Computer persönlich

Echtzeit-Betriebssysteme für Mikrocomputer

Grundlagen, Anwendungen, Systementwurf, Programmierpraxis von Realzeit-Multitasking-Betriebssystemen in der Praxis

Dr. Clemens Schmidt
Dietmar Albrecht

Echtzeit-Betriebssysteme für Mikrocomputer

Dr. Clemens Schmidt
Dietmar Albrecht

Echtzeit-Betriebssysteme für Mikrocomputer

Grundlagen, Anwendungen,
Systementwurf,
Programmierpraxis von
Realzeit-Multitasking-Betriebssystemen
in der Praxis

Markt & Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Schmidt, Clemens:
Echtzeit-Betriebssysteme für Mikrocomputer : Grundlagen, Anwendungen, Systementwurf, Programmierpraxis von Realzeit-Multitasking-Betriebssystemen in d. Praxis /
Clemens Schmidt ; Dietmar Albrecht. — Haar bei München : Markt-und-Technik-Verlag, 1984.
(Computer persönlich)
ISBN 3-922120-84-9
NE: Albrecht, Dietmar:

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

ISBN 3-922120-84-9

© 1984 by Markt & Technik, 8013 Haar bei München
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Eimannsberger, München
Printed in Germany

Vorwort

Das Buch will dem Leser praktische Erfahrung für den Entwurf und die Realisierung von Realzeitsystemen vermitteln. Dabei ist eine wesentliche Voraussetzung das Verständnis von Betriebssystemen, die es erst ermöglichen, eine effiziente Software auf einer leistungsfähigen Hardware einzusetzen.

Der Bezug zur Praxis war in jedem Teil des Buches wichtigstes Anliegen. Durch den kompletten Ausdruck des Quell-Codes des Realzeit-Multitasking-Betriebssystems ROSY (Realtime Operating System) sowie eines beispielhaften Anwenderprogramms wird der Leser in die Lage versetzt, die geschilderten Beispiele nicht nur vom Prinzip her, sondern im Detail nachzuvollziehen. Dadurch wird die wesentliche Voraussetzung für die praktische Verwertbarkeit der vermittelten Information geschaffen.

Die meisten Hardware-Beispiele werden am weitestverbreiteten 16-Bit Mikroprozessor, dem Intel 8086 behandelt. Entsprechend werden die Software Beispiele an der höheren Programmiersprache PL/M-86 erläutert, die für den 8086 Prozessor stärkste Verbreitung gefunden hat.

Es wird zuerst eingegangen auf Realzeit-Multitasking Betriebssysteme, wobei sowohl die Grundlagen als auch die praktische Realisierung dargelegt werden. Anschließend wird sowohl der Hardware- als auch der Softwaresystementwurf behandelt. Im letzten Teil wird beschrieben, wie Anwendersoftware entwickelt wird, die unter einem Realzeit-Multitasking-Betriebssystem ablaufen soll. Dabei werden sowohl wirkungsvolle grafische Hilfsmittel als auch die programmtechnische Realisierung von Prozessen erklärt.

Wir hoffen, mit dem Buch die Entwicklung zeitgemäßer Hard- und Software zu fördern.

Wir danken Herrn Bernhard Schaffer für seine Ermutigung und Unterstützung während der Entstehung des Buches. Wir danken Herrn Eberhard Knorpp für viele Diskussionen und kritische Anregungen.

Den Firmen Intel Semiconductor, Dr. Keil GmbH, Motorola, Scientific Control Systems GmbH, Siemens AG und Zilog danken wir für die freundliche Unterstützung.

Unser besonderer Dank gilt Herrn Klaus Kammerer für die gründliche Durchsicht großer Teile des Manuskripts. Seine vielen Anregungen und ständige, kritische Diskussionsbereitschaft haben wesentlich zum Gelingen des Buches beigetragen.

Frau Waltraud Albrecht danken wir für die Erstellung größerer Teile des Manuskriptes sowie vieler Zeichnungen.

Dem Markt&Technik Verlag danken wir für die stets vorbildliche Zusammenarbeit.

Einleitung

Realzeitsysteme sind solche, in denen das System in einer maximalen vorgegebenen Zeit auf äußere Einwirkungen reagiert haben muß. Die Anforderungen, die an diese Systeme gestellt werden, erfordern den Einsatz angemessener Hard- und Software.

Dabei werden zunehmend auch bei kleineren Anwendugnen Multitasking Betriebssysteme eingesetzt, die die quasi-gleichzeitige Bearbeitung mehrerer Probleme erlauben. Um diese wirkungsvollen Organisationsprogramme mit Erfolg benutzen zu können, muß man sich geeigneter Entwurfs-, Programmier- und Testmethoden bedienen. Obwohl für alle gängigen Mikrocomputer Realzeit-Betriebssysteme im Handel sind, ist es oft wünschenswert, die Systemsoftware nach eigenen, speziellen Anforderungen zu gestalten.

Im ersten Teil des Buches werden die grundlegenden Funktionsweisen von Realzeit-Multitasking-Betriebssystemen vermittelt. Ganz zu Anfang (Kap. 1.1) wird ein Überblick über die Gesamtfunktion eines Betriebssystems gegeben. Die Begriffe 'Realzeitfähigkeit' sowie 'Multitasking' werden in den Abschnitten 1.2 und 1.3 erläutert. Das 2. Kapitel bildet den Hauptabschnitt des ersten Teils. Dort wird auf den sog. Betriebssystemkern eingegangen, der die eigentliche Ablaufsteuerung bildet. Dieser hat für das Funktionieren von Realzeitsystemen herausragende Bedeutung.

Im zweiten Teil werden praktische Betriebssystemkerne erläutert. Den Hauptteil bildet der Betriebssytemkern 'ROSY', dessen vollständiges Compilerlisting in Anhang 1 abgedruckt ist. Kurzbeschreibungen verschiedener käuflicher Betriebssystemkerne runden diesen Teil ab.

Der dritte Teil behandelt Aspekte des Systementwurfes (Hardware und Software), die wichtig sind, wenn man mit einem Realzeit-Multitasking-Betriebssystem arbeitet. Hierunter fällt der für den Anwender zentrale Punkt, nach welchen Gesichtspunkten das Gesamtproblem in parallel ablaufende Teilprobleme aufgefächert werden soll, um ein möglichst überschaubares, stabiles Programmsystem zu erhalten. Ein Abschnitt ist Softwareproblemen für Multiprozessorsysteme gewidmet.

Der vierte Teil beschreibt praktische, erprobte Vorgehensweisen bei der Erstellung von Anwendersoftware, die unter einem Realzeit-Multitasking-Betriebssystem abläuft. Hier wird besonders eingegangen auf die Praxis der Zustands-Anreiz Programmierung.

Die beiden Anhänge bieten das erwähnte Compilerlisting des Betriebssystemkerns 'ROSY', sowie eines Anwenderprogramms, an dem die beschriebenen Vorgehensweisen praktisch nachvollzogen werden können.

Inhaltsverzeichnis

Erster Teil: Grundlagen und Arbeitsprinzipien 15

1. Betriebssysteme - Begriffsabgrenzungen 17
   1.1 Funktionsumfang eines Betriebssystems 18
      1.1.1 Betriebssystem-Kern 19
      1.1.2 Ein-/Ausgabe-System 20
      1.1.3 Dienstprogramme 24
      1.1.4 Kommunikationssystem (Transportsystem) 25
      1.1.5 Testsystem 33
         1.1.5.1 Offline-Testsystem 33
         1.1.5.2 Online-Testsystem 37
   1.2 Ein-Benutzer-, Mehr-Benutzer-Systeme 40
   1.3 Realzeit-Fähigkeit 44

2. Betriebssystem-Kern 47
   2.1 Prozesse 47
      2.1.1 Prozeßzustände 50
      2.1.2 Erzeugen und entfernen von Prozessen 52
   2.2 Warteschlangen 53
      2.2.1 FIFO Prinzip 54
      2.2.2 LIFO Prinzip 56
   2.3 Speicher-Pool-Verwaltung 57
      2.3.1 Pool-Elemente 60
      2.3.2 Verkettete Listen 64
      2.3.3 Verwaltung freier Poolelemente in einer Queue 70
      2.3.4 Verwaltung freier Poolelemente in einer Liste 70
   2.4 Stack-Verwaltung 72
      2.4.1 Funktionsweise des Stack 72
      2.4.2 Der Stack im Betriebssystem 75
      2.4.3 Stack-Zuweisung 78
   2.5 Prozeß-Kommunikation 79
      2.5.1 Briefkasten-Prinzip ("Mailbox") 80
         2.5.1.1 Versenden von Nachrichten 82
         2.5.1.2 Empfang von Nachrichten 83
      2.5.2 Warten auf Unterbrechung 84
      2.5.3 Semaphore, kritische Region 85
   2.6 Prozeß-Synchronisation 86
      2.6.1 Prioritäten 86
      2.6.2 Veränderung von Prozessen 88
      2.6.3 Vertagung, Fortsetzung von Prozessen 89

2.6.4 Verwaltung der Prozeß-Synchronisation 90
2.6.4.1 Process-Control-Block 91
2.6.4.2 Process-Scheduler 92
2.7 Zeitverwaltung 93
2.7.1 Zeitglieder (Timer) 94
2.7.2 Organisation der Zeitverwaltung 96
2.7.2.1 Feste Timerzuordnung 96
2.7.2.2 Variable Timerzuordnung 98
2.7.2.2.1 Doppelt verkettete Listen 102
2.7.2.3 Zuweisung und Löschen von Zeitgliedern 103
2.7.2.4 Erreichbare Maximalzeiten 107
2.7.3 Genauigkeit des Timers 108
2.7.4 Echtzeit-Uhr 109
2.8 Real-Time-Process 112
2.9 Betriebssystemüberwachungen 113
2.10 Zusammenfassung 114

Zweiter Teil: Praktische Betriebssysteme 117

3. Anwendung auf einen exemplarischen Betriebssystem-Kern 119
3.1 Typische Benutzerschnittstelle eines Betriebssystem-Kerns 119
3.1.1 Warteschlangenverarbeitung 119
3.1.1.1 QUEUE, UNQUEUE 121
3.1.1.2 ENQUEUE, DEQUEUE 122
3.1.2 Vergabe von Poolelementen 124
3.1.2.1 GET_POOL 124
3.1.2.2 RELEASE_POOL 125
3.1.3 Prozeß-Kommunikation 126
3.1.3.1 SEND_EVENT 126
3.1.3.2 RECEIVE_EVENT 127
3.1.3.3 WAIT_INTERRUPT, SIGNAL_INTERRUPT, DELAY, CONTINUE 129
3.1.4 Prozeß-Synchronisation 129
3.1.4.1 PROCESS_SCHEDULER 129
3.1.4.2 SET_PRIORITY, RESET_PRIORITY 130
3.1.5 Zeitglieder zuweisen und entlassen 131
3.1.5.1 TIMER_ASSIGN 131
3.1.5.2 TIMER_RELEASE 132
3.1.5.3 Teilqualifizierte Zeitglieder 134

3.2 Systemstart 135
3.2.1 Laden des Betriebssystems 135
3.2.2 Initialisierung des Betriebssystems 136
3.2.2.1 Speicherinitialisierung 136
3.2.2.2 Interrupt-Tabelle 136
3.2.2.3 Process-Load-Table 136
3.2.2.4 Stack 137
3.2.3 Start der Prozesse 137
3.3 Real-Time-Process 138
3.4 Das Betriebssystem ROSY 139
3.4.1 Leistungsmerkmale von ROSY 140
3.4.2 Benutzerschnittstelle von ROSY 140
3.4.3 Systemgenerierung mit ROSY 141
4. Kurzbeschreibung verschiedener Betriebssystem-Kerne 143
4.1 iAPX 86/30 Betriebssystem-Prozessor kompatibel mit iRMX 86 144
4.2 RMOS, MTOS 86 145
4.3 RMS 68K 146
4.4 ROSY 147
4.5 ZRTS 8000 148

Dritter Teil: Systementwurf 149

5. Systementwurf im Hinblick auf die Umgebung eines Realzeit-Betriebssystems - Hardware Aspekte 151
5.1 Minimalanforderungen eines Realzeit-Betriebssystems an die Hardware 151
5.1.1 RAM-Speicher 151
5.1.2 ROM-Speicher 155
5.1.3 Zeitgeber 155
5.1.4 Unterbrechungen (Interrupts) 158
5.1.4.1 Maskierbare Unterbrechungen 158
5.1.4.2 Nicht-maskierbare Unterbrechungen 162
5.1.5 Zusammenfassung 163
5.2 Sonstige Hardware-Eigenschaften kleiner Systeme 164
5.2.1 Anschluß einer Datensichtstation 164
5.2.2 LED-Anzeigen 165
5.2.3 Watchdog 167
5.2.4 Realisierungsbeispiel 171

5.3 Ein-/Ausgabe-System 174
5.3.1 Eingabe von Sensoren unter Realzeitgesichtspunkten 175
5.3.2 Serielle Schnittstellen 176
5.3.3 Geräte mit hohen Datenraten 178
5.4 Hardware-Eigenschaften komplexer Systeme 179
5.4.1 Monoprocessing - Multiprocessing 179
5.4.2 Multiprocessing - Feste Kopplung 182
5.4.3 Multiprocessing - Lose Kopplung 184
5.4.4 EWSD - ein Beispiel für ein Multiprozessorsystem 185
5.4.5 Redundanz 188
6. Systementwurf in Hinblick auf die Umgebung eines Realzeit-Multitasking-Betriebssystems Software Aspekte 190
6.1 Allgemeine Kriterien zur Software-Systemstruktur 190
6.2 Realisierung der Software-Struktur 191
6.2.1 Wahl des Betriebssystems 191
6.2.2 Analyse der Aufgabenstellung - Aufteilung in Prozesse 194
6.2.2.1 Ein Beispiel 194
6.2.2.2 Folgerungen aus dem Beispiel 203
6.2.3 Prozeß-Prioritäten 206
6.2.3.1 Bewertungskriterien 206
6.2.3.2 Änderung von Prioritäten im Betrieb 211
6.2.3.3 Zeitliche Verschachtelung der Prozesse 212
6.2.3.4 Zusammenfassung "Prioritäten" 213
6.2.4 Realzeit-Aspekte 214
6.2.4.1 Interrupts 214
6.2.4.2 Einfluß des Betriebssystems auf die Systemdynamik 216
6.2.5 Prozeß-Überwachungen 217
6.2.5.1 Zeitüberwachungen 217
6.2.5.2 Plausibilitätskontrollen 218
6.3 Die Datenbasis 218
6.3.1 Zugriff auf gemeinsame Daten 219
6.3.2 Permanente Daten 221
6.3.3 Transiente Daten 221
6.3.4 Semipermanente Daten 221
6.4 Multiprocessing 223
6.4.1 Funktionsteilung ("Function Sharing") 223
6.4.2 Lastteilung ("Load Sharing") 224

Vierter Teil: Softwareentwicklung für Anwenderprozesse 231

7. Softwareentwicklung für Anwenderprozesse in einer Realzeit-Multitasking-Betriebssytem-Umgebung 233
7.1 Zustands-Anreiz-Programmierung 233
7.1.1 Zustände von Prozessen 233
7.1.2 Anreize an Prozesse 237
7.1.3 Folgerungen aus dem Beispiel 238
7.1.4 "Step Tables" 240
7.2 Grafische Hilfsmittel 242
7.2.1 Struktogramme 242
7.2.2 Zustands-Anreiz-Diagramme (SDL-Diagramme) 245
7.2.2.1 SDL-Symbole 246
7.2.2.2 Beispiele 248
7.3 Die Organisation von Prozessen 256
7.3.1 Prozeß-Hochlauf 256
7.3.2 Die Organisation der Endlosschleife 257
7.3.2.1 Klassifikation von Meldungen 258
7.3.2.2 Verteilung der Anreize an die Anwenderprogramme 260
7.3.3 Teilnehmerindividuelle Meldungen 264
7.3.4 Priorisierung von Meldungen 265
7.3.4.1 Prozeßinterne Meldungen 265
7.3.4.2 Priorisierung von Meldungen zwischen verschiedenen Prozessen 267
7.3.5 Verzögern von Meldungen 269
7.3.6 Verwaltung von Prozeß-Zuständen 270

Anhang 1: Compilerlisting des Betriebssystem-Kerns 'ROSY' (REAL TIME OPERATING SYSTEM) 273

Anhang 2: Compilerlisting des Anwenderprozesses 'ÜBERWACHUNG' 303

Stichwortverzeichnis 315

Erster Teil:

Grundlagen und Arbeitsprinzipien

Enter Tab

1. Betriebssysteme - Begriffsabgrenzungen

Die Entwicklung der Mikroelektronik geht dahin, in immer kompaktere und billigere Hardware immer mehr Leistung zu packen. Diese Leistungsfähigkeit beruht einerseits auf höherintegrierten und schnelleren Rechner- und Peripheriebausteinen, andererseits auf Programmen, die die vorhandene Hardware besser ausnutzen. An einem Beispiel sei dies verdeutlicht:

Stellen Sie sich vor, ein Programm oder Programmteil hat die Aufgabe, einen bestimmten Zeitwert zu überwachen. Ein mikrocomputergesteuerter Telefonapparat ("Telefoncomputer") soll auf den Druck einer Taste des Ziffernblockes hin die entsprechende Ziffer wählen. Das Öffnen und Schließen des Wahlkontaktes muß in genauen, von der Post vorgeschriebenen Zeiten erfolgen. Der Programmierer kann die gewünschte Zeit mittels Software auf einfachste Weise durch Setzen und Herunterzählen einer Zählervariablen erzeugen. Diese Methode funktioniert bestens, solange im Gerät sonst keine Forderung besteht, noch weitere Zeiten zu überwachen. Stellen Sie sich aber weiter vor, es sei vom Käufer des Gerätes gefordert, man solle auf Tastendruck jederzeit auf einer LCD-Anzeige bestimmte Informationen, wie Uhrzeit oder Gebühren, abfragen können, und zwar soll z.B. die Information 20 Sekunden stehen bleiben und dann verlöschen.

Das Gerät soll außerdem überwachen, ob eine Taste klemmt, d.h. ob eine Taste nach ihrer Betätigung nach spätestens 10 Sekunden wieder losgelassen wird.

Der Programmierer ist nun in der schwierigen Lage, drei Zeitabläufen überwachen zu müssen, wobei er damit rechnen muß, daß alle drei Zeitkontrollen gleichzeitig und unabhängig voneinander laufen müssen.

Mit der einfachen Zeitschleife ist dieses Problem nicht mehr zu lösen oder nur noch mit Tricks, die das ganze Programm schwerfällig und unflexibel machen.

Der Programmierer benötigt zur Lösung des Problems ein Zeitprogramm, das er aufruft, wenn er eine Zeit zu überwachen hat, und das nach Ablauf der entsprechenden Zeit das richtige Bearbeitungsprogramm anstößt, wobei aber das Zeitprogramm nicht den Rechner blockieren darf.

Dieser gewünschte Komfort wird von einem modernen Realzeit-Betriebssystem geboten.

Mit diesem einführenden Beispiel sollte eigentlich nur eines gezeigt werden: Mit steigender Anforderung an einen Rechner, müssen die vorhandenen Betriebsmittel (hier die Zeit, die der Rechner für ein Problem aufwenden darf) ökonomischer eingeteilt werden. Dazu helfen Betriebssysteme.

Eine weitere Aufgabe von Betriebssystemen ist es, die Benutzung der Rechenanlage für den Menschen zu vereinfachen. Als Beispiel sei der Gebrauch von Übersetzungsprogrammen, wie Assembler, angeführt. Es wird also die These aufgestellt:

Betriebssysteme sind Dienstleistungsprogramme, die die Verwaltung der vorgegebenen Betriebsmittel einer Rechenanlage erleichtern und die Benutzung der Rechenanlage vereinfachen.

In den folgenden Unterkapiteln wird versucht, einige Beispiele von wichtigen Komplexen eines Betriebssystems zu geben.

1.1 Funktionsumfang eines Betriebssystems

Aus welchen Teilen setzt sich ein Betriebssystem zusammen und welche Funktionen erfüllen diese einzelnen Teile?

Ein Betriebssystem ist in Schichten gegliedert (Vgl. Abb. 1-1). Man spricht üblicherweise von einem Betriebssystem-Kern und einer Betriebssystem-Schale. Das Betriebssystem wird vom "Anwenderprogramm" benutzt. Das Anwenderprogramm führt die eigentlichen Funktionen des Gerätes oder der Steuerung, die z.B. in einem Pflichtenheft niedergelegt sind, aus.

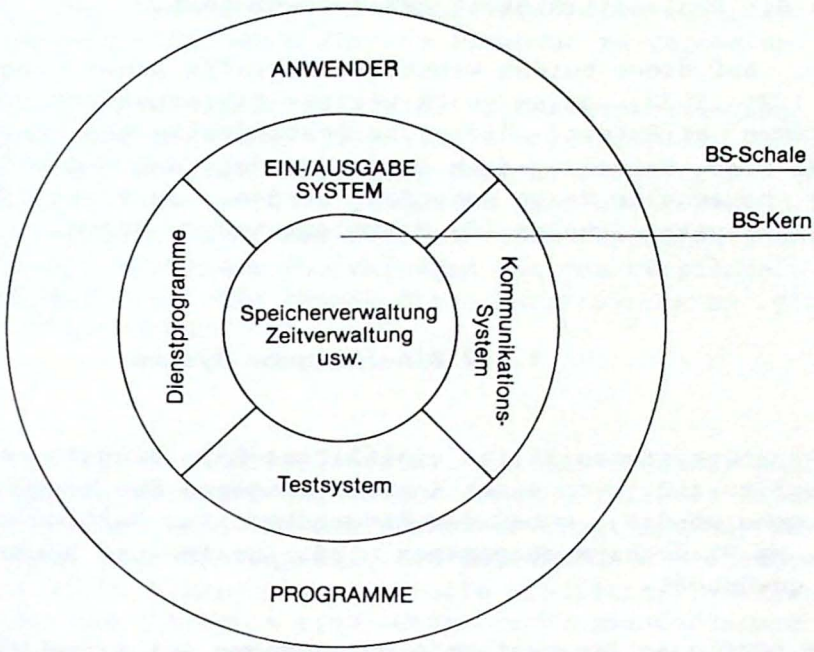

Abb. 1-1: Schichtung des Bertriebssystems

1.1.1 Betriebssystem-Kern

Der Betriebssystem-Kern ist diejenige Einheit, auf die das gesamte Betriebssystem abgemagert werden kann, so daß es noch sinnvoll von einem Anwenderprogramm nutzbar ist. In diesem Teil werden die Grundfunktionen des Betriebssystems, wie Speicherverwaltung, Zeitverwaltung, Datenverwaltung, ausgeführt und es werden die Abläufe der einzelnen Rechenprozesse geregelt usw.

Alle diese Schlagworte werden in weiteren Kapiteln ausführlich erläutert werden, da der Betriebssystem-Kern einen zentralen Bestandteil des Betriebssystems bildet. Im Betriebssystem-Kern ist ein beachtlicher Teil der möglichen Leistungsfähigkeit eines Rechnersystems verborgen. Hier wird z.B. festgelegt, ob es sich um ein Ein-Benutzer oder Mehr-Benutzer ("multitasking") Be-

triebssystem handelt (s. 1.2). Weiter legt der Betriebssystem-Kern die Realzeitfähigkeit des Systems fest.

Bevor auf diese beiden wichtigen Begriffe näher eingegangen wird (s. 1.2, 1.3), seien noch weitere Eigenschaften von Betriebssystemen erläutert, die nicht Bestandteile des Betriebssystem-Kerns sind. Es soll jedoch betont werden, daß diese Teile hier in mehr pauschaler Weise behandelt werden, um einen Eindruck von Betriebssystem-Schalen für Mikrocomputer zu geben.

1.1.2 Ein-/Ausgabe-System

In Rechnersystemen fallen vielfältige Ein-/Ausgabeoperationen an. So sollen z.B. von einer Konsole Eingaben des Menschen entgegengenommen werden, wobei das Eingegebene zur Kontrolle gleich wieder am Bildschirm ausgegeben wird. Listen und Ausdrucke werden oft gewünscht.

Sehr häufig sollen bestimmte Datenmengen auf einem Massenspeicher wie Plattenspeicher, Magnetband oder Floppy Disk festgehalten werden. Bei Bedarf will man dann jederzeit wieder auf die Daten zugreifen können.

Betriebssysteme sorgen dafür, daß der Benutzer diese Daten in einer für ihn bequemen Weise ansprechen kann, nämlich in Form von Dateien, auf die er dann unter einem bestimmten Namen zugreift. Der Benutzer muß sich also nicht darum kümmern, an welcher Stelle auf dem Datenträger seine Daten liegen. Das Betriebssystem übernimmt diese Verwaltungsaufgaben. Eine Datei kann z.B. auf einer Floppy Disk über viele kleine Bereiche verstreut abgelegt sein. Das Betriebssystem gewährleistet, daß immer auf die richtige Stelle auf dem Datenträger zugegriffen wird.

Ein-/Ausgabeoperationen sollen für den Anwender möglichst komfortabel handzuhaben sein. Das bedeutet z.B., daß der Anwender für alle Geräte eine einheitliche Schnittstelle benutzen kann, wo das Ein-/Ausgabemedium als Parameter angegeben ist. So könnten etwa mit dem Prozeduraufruf WRITE Daten in eine Datei auf einer Diskette geschrieben werden, oder an eine Datensichtstation ausgegeben werden. Ebenso ist ein Betriebssystemprogramm READ denkbar, das Eingaben von allen möglichen Geräten erlaubt. Der Anwender

will sich nicht damit befassen müssen, wie die Details der Gerätebedienung vor sich gehen. Das Betriebssystem hat also die Aufgabe, die benutzte Hardware für den Anwender zu verdecken.

Ein sehr häufig vorkommender Fall sind Ein-/Ausgabevorgänge an einer Datensichtstation über eine V.24 Schnittstelle, die von einem seriellen Schnittstellenbaustein bedient wird (z.B. USART 8251 von Intel). Daher wird im folgenden kurz auf einige Besonderheiten eingegangen, die beim Betrieb einer seriellen Schnittstelle auftreten. Für diese Ausführungen ist das Verständnis von Kapitel 2 erforderlich. Sie können daher bedarfsweise an dieser Stelle überschlagen werden.

Ausgabe

Wird ein Byte an einen USART abgegeben, muß das Ende der Ausgabe (Parallel-Seriell-Wandlung) abgewartet werden, bis das nächste Byte übergeben werden kann. Um den Prozessor nicht mit der Wartezeit zu belasten, ist es sinnvoll, die Ausgabe unterbrechungsgesteuert ablaufen zu lassen. Der serielle Schnittstellenbaustein kann dazu nach jeder Ausgabe eine Unterbrechungsanforderung an die CPU abgeben; diese übergibt dann das nächste Byte zur Ausgabe.

Wie erwähnt, soll für den Anwender dieser Vorgang transparent ablaufen. Um dieses Ziel zu erreichen, ist es sinnvoll, zwischen Anwenderprozeß und Interrupt-Routine einen eigenen Ausgabeprozeß zu schalten, den man dann als Ein-/Ausgabe-System der Betriebssystem-Schale bezeichnen würde. So kann auf einfache Weise das Problem gelöst werden, daß mehrere Anwender gleichzeitig auf das periphere Medium zugreifen können.

Eine mögliche Realisierung der Ausgabe soll kurz skizziert werden (vgl. Abb. 1-2):

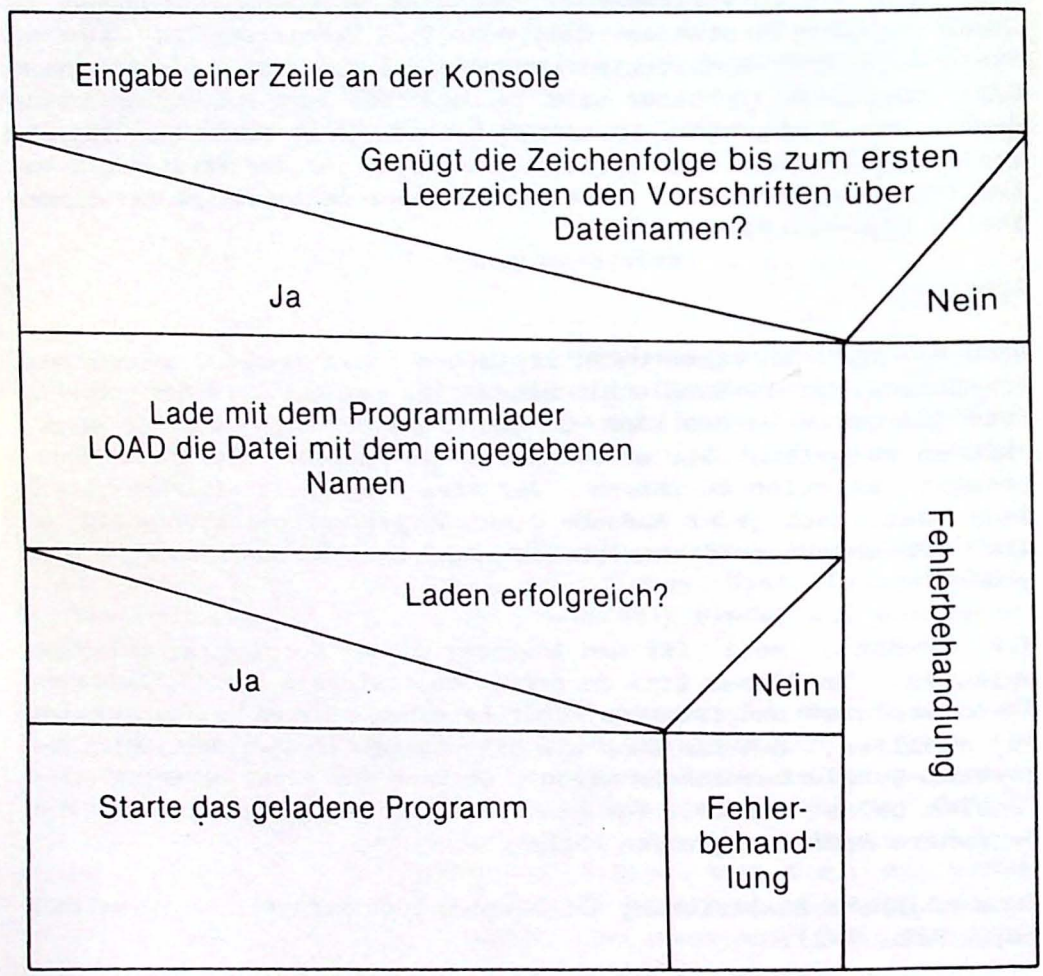

Abb. 1-2: Ausgabe an Datenstation

Der Ausgabeprozeß kopiert die auszugebenden Zeichen des Ausgabe-Interruptprogramms zusammen mit der Zahl der Zeichen in ein festes Speicherfeld. Anschließend wird am USART ein Nullzeichen ausgegeben, was am Sichtschirm nichts bewirkt (z.B. ASCII_NULL). Dieses Blindzeichen hat lediglich den Sinn, einen ersten Interrupt vom USART zu erzeugen. Damit wird das Ausgabe-Interruptprogramm angestoßen, das dann alle weiteren Zeichen aus der Ausgabeliste an den USART abgibt und mit jedem Interrupt nach erfolgter Ausgabe neu angestoßen wird. Das wiederholt sich solange, bis das Ausgabe-Interruptprogramm erkennt, daß die Liste vollständig ausgegeben ist. Danach macht das Interruptprogramm dem Interruptprozeß eine Mitteilung (z.B. mit SEND_EVENT, s. 2.5.1.1, 3.1.3), daß das Ausgabefeld wieder frei ist. Werden während der laufenden Ausgabe neue Ausgaben vom Ausgabeprozeß angefordert, muß dafür gesorgt werden, daß der neue Text nicht die alte noch aktuelle Ausgabeliste überschreibt. Das kann z.B. so geschehen, daß der Interruptprozeß bis zum Eintreffen der Meldung "Ausgabeliste frei" vom Interruptprogramm die Ausgabeanforderungen zwischenspeichert.

Das beschriebene Verfahren hat den Vorteil, daß die Realzeitebene, also das Interruptprogramm, wenig zu tun hat. Es muß lediglich den Index im Ausgabepuffer verwalten und hochzählen, aber keine komplizierten, zeitaufwendigen Adressierungen ausführen. (In welchen Zeitabständen die Ausgabeunterbrechungen kommen, zeigt für verschiedene Datenraten Tabelle. 5-1, s.5.3.2). Der Sinn der beschriebenen Konstruktion ist der, daß das Interruptprogramm möglichst kurz läuft (vgl. 6.2.4.1).

Eingabe

Die Eingabe von der Tastatur der Datensichtstation ist zeitlich unkritisch, da auch bei schneller Eingabe maximal alle 100ms ein Eingabeinterrupt kommen kann. Hier kann man es sich erlauben, nach jedem eingegebenen Zeichen mit einem SEND_EVENT (s. 2.5.1.1, 3.1.3) das Zeichen dem zuständigen Prozeß zu übermitteln, der dann die weitere Auswertung vornimmt. Denkbar ist aber auch der Eintrag in eine Liste, die bis zur Eingabe eines Endezeichens (z.B. Wagenrücklauf) vollgeschrieben wird und für neue Eingaben solange durch ein Merkbit verrriegelt wird, bis der zuständige Prozeß die Liste abgearbeitet hat.

Hier soll nicht weiter auf Ein-/Ausgabe-Systeme eingegangen werden. Einige weitere, hardwarebezogene Aspekte dieses Problemkreises werden in Kapitel 5.3 behandelt.

1.1.3 Dienstprogramme

Dieser Punkt soll hier nur kurz gestreift werden, da er für die Struktur von Realzeit-Betriebssystemen wenig Bedeutung hat.

Unter Dienstprogrammen versteht man eine Anzahl von Programmen, die das Betriebssystem dem Benutzer für häufig vorkommende Aufgaben zur Verfügung stellt. Es ist oft aber gar nicht eindeutig zu sagen, inwieweit die Dienstprogramme überhaupt zum Betriebssystem zählen. Für die Programmentwicklung z.B. sind die wichtigsten Dienstprogramme:

- EDITOR — zur Texterstellung und -Manipulation, oft zur Erstellung von Quellprogrammen benutzt.
- ASSEMBLER — zur Übersetzung eines in Assemblercode geschriebenen Programmes in Maschinencode.
- COMPILER — zur Übersetzung eines in einer höheren Programmiersprache (z.B. PASCAL, PL/M, CHILL) geschriebenen Programmes in Maschinencode.
- LINKER — zum Binden einzelner Programmteile zu einem Gesamtsystem und anschließendem Laden in den Hauptspeicher.

Der Test der Programme auf logische Korrektheit wird dann mit dem Testsystem (s. 1.1.5) vorgenommen. Umfang und Komfort der Testsysteme sind sehr unterschiedlich, sie reichen von einfachen Monitorprogrammen bis zu aufwendigen On-Line Testsystemen.

Die üblichen Stationen der Programmerstellung und durch den Test bedingten Programmänderungen sind (s. Abb. 1-3):

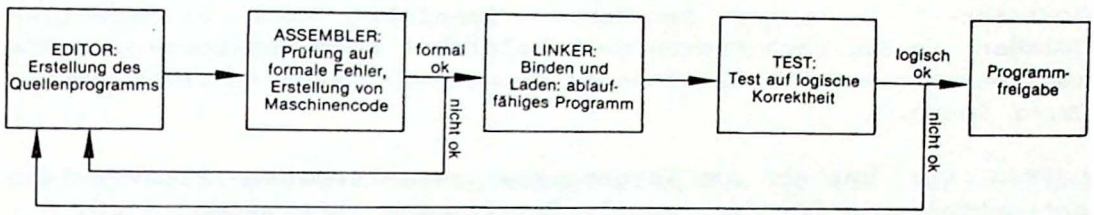

Abb. 1-3: Stationen der Programmentwicklung

Die beiden folgenden Unterkapitel "Transportsystem" und "Testsystem" setzen Kenntnisse von Begriffen des Betriebssystem-Kerns voraus. Sowohl Transportsystem als auch Testsystem gehören nicht zum Betriebssystem-Kern, sondern zur Betriebssystem-Schale, daher werden sie an dieser Stelle des Buches eingeordnet.

1.1.4 Kommunikationssystem (Transportsystem)

Lassen Sie uns wieder mit einem anschaulichen Beispiel beginnen:

Fritz Müller aus Hamburg schreibt Egon Schulze aus Freiburg einen Brief. Er steckt ihn in einen Umschlag, schreibt auf diesen die Anschrift von Schulze und steckt ihn in den Briefkasten. Dann geht er nach Hause und ist sicher, daß sein Freund Schulze spätestens in zwei Tagen seine Nachricht liest.

An diesem Beispiel lassen sich bei näherer Betrachtung einige wichtige Grundprinzipien von Rechner-Kommunikationssystemen ablesen.

Warum z.B. steckt Müller den Brief in einen Umschlag? Er könnte ja die Anschrift auch einfach so auf den Brief schreiben. Sie werden sagen, die Post verlangt nun mal ein Kuvert (das Briefgeheimnis könnte auch gewahrt werden, wenn der Brief chiffriert wird).

Versuchen wir, den Weg des Briefs zu verfolgen:

Brief schreiben--> Umschlag--> Briefkasten--> Postamt--> Postsack--> Postwagen der Bahn--> Bahnfahrt nach Frankfurt--> Umladen in Zug nach Freiburg--> Bahnfahrt nach Freiburg--> Postsack aussortieren--> Briefkasten Schulze--> Umschlag entfernen--> Brief lesen.

Lassen wir uns nun den Vorgang aus verschiedenen Blickwinkeln betrachten, die für die jeweils Beteiligten interessant sind:

Müller	Schulze
Brief schreiben und	Brief lesen und
in Kuvert stecken und adressieren	Kuvert öffnen und
Brief in Briefkasten	Briefkasten leeren

Postamt Hamburg	Postamt Freiburg
Briefkasten leeren	Brief in Briefkasten
Brief sortieren und in Postsack	Brief aus Postsack
Brief zur Bahn	Brief von Bahn

Bundesbahn

Hamburg	Frankfurt	Freiburg
Postsack in Zug nach Frankfurt	Postsack umladen	Postsack ausladen

---

Wie man sieht, haben die drei Beteiligten, nämlich

1. Müller und Schulze
2. Post
3. Bahn

völlig unterschiedliche Interessen an dem Brief. Müller und Schulze interessiert eigentlich nur der Inhalt; sobald Schulze den Brief aus dem Kasten nimmt, hat das Kuvert seine Schuldigkeit getan, es wandert in den Müll.

Müller bedient sich der Bundespost als Dienstleistungsunternehmen. Er muß sich an die vorgeschriebenen Regeln halten, damit sein Brief befördert wird. Welchen Weg sein Brief genau nimmt, weiß er nicht und es interessiert ihn auch nicht, ebensowenig ob er mit der Bahn, mit dem Hubschrauber oder mit Rheinkähnen transportiert wird, wenn er nur rechtzeitig und sicher ankommt.

Dasselbe gilt für Schulze, auch ihn interessiert nur die eigentliche Nachricht, der Inhalt des Briefes.

Ganz anders die Bundespost. Der Inhalt der Korrespondenz interessiert sie nicht. Der Post ist es egal, ob der Postsack einmal oder zehnmal umgeladen wird. Der Postsack wird am Zielort, wenn auch nicht in den Müll geworfen, so doch zur Seite gelegt, nachdem die Briefe ausgeschüttet sind. Auch der Postsack ist genormt, ähnlich wie der Briefumschlag, damit er überhaupt zur Beförderung zugelassen wird. So ist eine effiziente und sichere Beförderung und Umladung möglich. Es ist der Post aber schon nicht mehr egal, wie der Brief befördert wird. Ein Expressbrief etwa bedarf besonderer Zustellungsverfahren.

Wieder anders die Bahn. Sie ist ausschließlich am Postsack interessiert. Sie hat dafür zu sorgen, daß er den aufgedruckten Zielort sicher erreicht. Sie muß also selbständig dafür sorgen, daß der Postsack in Frankfurt umgeladen wird. Und sollte der Zug von Frankfurt nach Freiburg entgleisen, muß sie für einen Ersatzweg sorgen. Will die Post Briefe per Bahn in Säcken befördern, muß auch sie bestimmte Regeln (Aussehen, Größe, Beschriftung der Postsäcke) einhalten.

Abstrahieren wir etwas:

1. Die Übermittlung der Nachricht erfolgt in verschiedenen Schichten, wobei jede Schicht einen genau definierten Zuständigkeitsbereich hat.

2. Jeder an der Kommunikation Beteiligten, zwischen Start- und Zielort, verkehrt nur mit Partnern seiner eigenen Schicht.

3. Die Kommunikation innerhalb einer Schicht erfolgt nach genormten Regeln.

4. Höhere Schichten benutzen niedrigere Schichten als Dienstleistungen.

5. Jede Schicht benutzt ihre eigene Verpackung für die Nachricht.

6. Jede Schicht handelt in ihrem Aufgabenbereich vollverantwortlich.

7. Die übergeordnete Schicht darf annehmen, daß der gebotene Dienst gesichert ist.

In unserem Beispiel könnten wir drei Schichten folgendermaßen festlegen:

- Schicht 3: Benutzerschicht (Schulze-Müller).
- Schicht 2: Ende-zu-Ende Übertragungsschicht (von Postkasten zu Briefkasten).
- Schicht 1: Transportschicht (Bahnnetz).

Ein wichtiger Begriff ist der des Protokolls. Das bedeutet, daß je zwei Partner derselben Schicht nur mit genau festgelegten, möglichst genormten Verfahrensweisen miteinander verkehren dürfen. In unserem Beispiel etwa wird der Brief am Startort gestempelt. Damit weiß die Post am Zielort, daß sie den Brief zustellen kann.

Lassen Sie uns nun ein analoges Beispiel aus dem Bereich der Kommunikation zwischen Rechnern anführen, aus dem das Prinzip der sog. "Offenen Kommunikationssysteme" deutlich wird:

Mehrrechnersysteme in "loser Kopplung" (s. 5.4.3) sind oft mit seriellen Datenleitungen verbunden (Abb. 1-4):

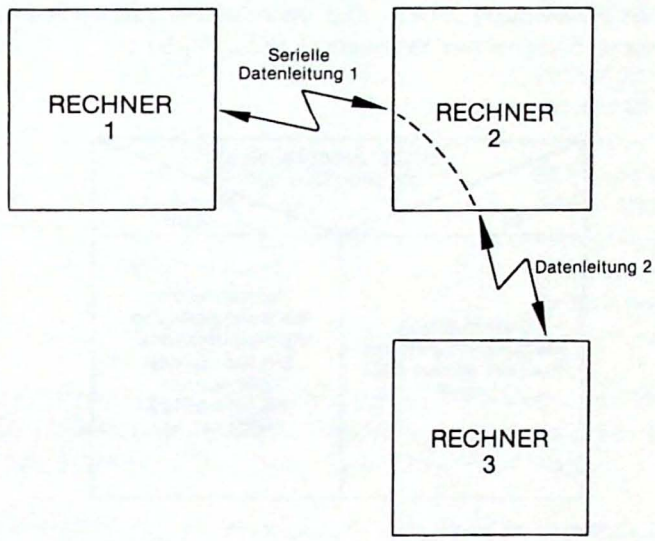

Abb. 1-4: Rechnernetzwerk in loser Kopplung

In jedem der drei Rechner sei ein Betriebssystem implementiert. Angenommen, ein Prozeß PROZESS_12 im Rechner 1 möchte eine Meldung (repräsentiert durch eine Zeigervariable POINTER) an einen Prozeß PROZESS_13 im gleichen Rechner senden. Dazu bedient er sich des Betriebssystem-Aufrufs, z.B.

```
SEND_EVENT(PROZESS_13, POINTER);
```

Diese Funktion ist im Betriebssystem-Kern von Rechner 1 enthalten, die Meldung bleibt innerhalb Rechner 1.

Im weiteren Verlauf des Programms wird es notwendig, daß PROZESS_12 im Rechner 1 eine Meldung an den Prozeß PROZESS_32 im Rechner 3 sendet. Der Anwender bedient sich des gleichen Betriebssystem-Aufrufes:

```
SEND_EVENT (PROZESS_32, POINTER);
```

Aufgabe des Betriebssystems ist es nun herauszufinden, ob dieser Prozeß in Rechner 3 läuft, und dann das Transportsystem anzustoßen, das die Meldung über die serielle Datenleitung transportiert, etwa nach folgendem Schema (Abb. 1-5):

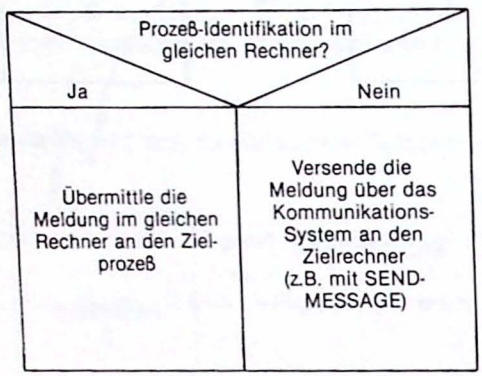

Abb. 1-5

Im sendenden Rechner erledigt ein Unterprogramm, z.B.

```
SEND_MESSAGE
```

den Transport der Mitteilung auf die Datenleitung. Im Empfangsrechner sorgt ein entsprechendes Programm

```
RECEIVE_MESSAGE
```

für Empfang und korrekte Zustellung der Sendung. Soll die Sendung, so wie in unserem Beispiel, an einen dritten Rechner durchgereicht werden, könnte ein entsprechendes Programm

```
SWITCH_MESSAGE
```

diese Funktion besorgen.

Die Analogie mit unserem Beispiel des Brieftransportes ist offensichtlich. Lassen Sie uns vergleichen:

Schicht	Brieftransport	Meldungstransport
Anwender	Müller->Schulze	PROZESS_12 -> PROZESS_32
Ende-zu-Ende	Post Hamburg -> Post Freiburg	Betr.System 1 -> Betr.System 3
Transport	Bundesbahn	Transp.System 1 -> Transp.System 3

PROZESS_12 im Rechner 1 will lediglich, daß der PROZESS_32 im Rechner 3 die Meldung sicher bekommt. Auf welche Art das Betriebssystem das bewältigt, ist dem Anwender egal.

Hat das Betriebssystem im Rechner 1 einmal festgestellt, daß die Meldung nach außerhalb geht, aktiviert es das Transportsystem mittels SEND_MESSAGE und verschickt die Meldung an Rechner 2, da zu Rechner 3 keine direkte Datenleitung besteht.

Das Transportsystem im Rechner 2 stellt lediglich fest, daß die Meldung Rechner 3 gehört und schickt sie mittels SWITCH_MESSAGE weiter.

Das Transportsystem im Rechner 3 (RECEIVE_MESSAGE) übergibt die Meldung dem Betriebssystem-Kern und dieser gibt sie schließlich dem Anwender.

Auf die Realisierungsdetails von Transportsystemen soll hier nicht eingegangen werden.

Aufbauend auf diesem Beispiel, ist der Begriff des "Offenen Systems" verständlich. In einem beliebigen Rechnernetzwerk sollen damit Anwender des einen Systems, Nachrichten in jedes der anderen Systeme übermitteln können. Dazu ist von ISO (=International Standards Organization) der Vorschlag gemacht worden, die Nachrichtenübermittlung auf sieben Schichten ablaufen zu lassen ("ISO Reference Model for Open Systems Interconnection"). Diese haben folgende Funktion:

- Schicht 7: Anwenderschicht.

- Schicht 6: Präsentationsschicht.
  Sie regelt die Darstellung der verwendeten Zeichen, z.B. ASCII-Code oder EBCDIC-Code.

- Schicht 5: Sitzungsschicht.
  Für die Dauer der Verbindung zweier Anwender besteht eine "Sitzung". Der Ablauf (Aufbau, Abbau etc.) wird hier geregelt.

- Schicht 4: Netzschicht.
  Zerteilt größere Datenmengen in kleinere Pakete und setzt sie am Zielort wieder richtig zusammen.

- Schicht 3: Verbindungsschicht.
  Stellt für die Datenpakete einen Weg durch das Netz zur Verfügung. Es ist nicht notwendig, daß für jedes Paket derselbe Weg genommen wird. ("Datagramme")

- Schicht 2: Transportschicht.
  Sorgt für den sicheren Transport innerhalb eines Wegabschnittes.

- Schicht 1: Physikalische Schicht.
  Sorgt für das korrekte Aussenden bzw. den richtigen Empfang der Daten auf der Leitung.

Die Schichten 1 bis 4 regeln transportspezifische Abläufe, die Schichten 5 bis 7 anwenderspezifische Abläufe.

Für den internationalen Datenaustausch ist die Normung der Protokolle von großer Wichtigkeit. Internationale Normen gibt es bis jetzt lediglich für Schicht 1 (z.B. CCITT-Norm X.21), Schicht 2 (z.B. HDLC = High Level Data Link Control), und Schicht 3 (z.B. X.25 Norm).

In unserem Beispiel entspricht die Ebene der Prozesse der Schicht 7, die Ebene des Betriebssystem-Kerns der Schicht 4, die Ebene des Transportsystems der Schicht 2. Die Behandlung der physikalischen Anschlüsse (z.B. steuerbare Leitungstreiber, Ein-/Ausgabesteuerports) geschieht auf Schicht 1. Die übrigen Schichten sind hier leer.

1.1.5 Testsystem

Auch für diesen Abschnitt wird die Kenntnis des Prozeßkonzeptes (Kapitel 2) vorausgesetzt.

Unter Testsystem versteht man die Gesamtheit der unter Betriebssystem-Aufsicht ablaufenden Hilfsmittel, die dem Programmierer helfen, logische Fehler in seinem Programm zu finden. Der Einsatz eines Testsystems setzt also voraus, daß

- die einzelnen Programmoduln fehlerfrei vom Compiler bzw. Assembler übersetzt sind,
- die Moduln zu einem Gesamtsystem vom Linker zusammengefügt wurden,
- dieses Programmsystem jederzeit ladbar (falls es z.B. von Floopy Disk geladen wird) und ausführbar ist.

Man unterscheidet zwischen Offline- und Online-Testsystemen.

1.1.5.1 Offline-Testsystem

Offline-Testsysteme sind solche, die einen Programmtest unter anderen als den im Normalbetrieb üblichen Bedingungen erlauben. Folgende Beispiele sollen den Begriff verdeutlichen.

Beispiel 1

Angenommen, es sei ein Gerät zu entwickeln, welches Eingaben von einer Tastatur zuläßt (z.B. der Telefoncomputer aus dem einführenden Beispiel). Einer der beteiligten Programmierer ist damit beauftragt, einen Tastenscanner zu programmieren, d.h. ein Programm, welches in periodischen Zeitabständen die Tastenzustände abfragt und im Falle einer Änderung (Druck oder Loslassen einer Taste) eine Aktion veranlaßt. Außerdem soll ein sog. "Last Look" realisiert werden. Das heißt, daß nach einer erkannten Änderung des Tastenfeldes eine bestimmte Zeit gewartet werden soll, um etwaiges Prellen der Tasten abzuwarten. danach wird der Zustand der Taste erneut abgefragt. Nur falls der Zustand der Taste

derselbe ist wie unmittelbar nach der Änderung, wird die Taste als gültig gewertet. Andernfalls wird angenommen, es handelte sich um eine Störung. Eine typische Wartezeit sind 20 Millisekunden. Bei diesem Vorgang handelt es sich also um ein softwaremäßiges Entprellen der Tastatur.

Der Programmierer realisiert das Programm als Prozeß, der die Tastenänderungen als 'Anreize' (vgl. 7.1.2) aufnimmt, den Last Look durchführt und falls keine Störung vorlag, eine Meldung an den weiterverarbeitenden Prozeß abgibt. Schematisch sähe das so aus (Abb. 1-6):

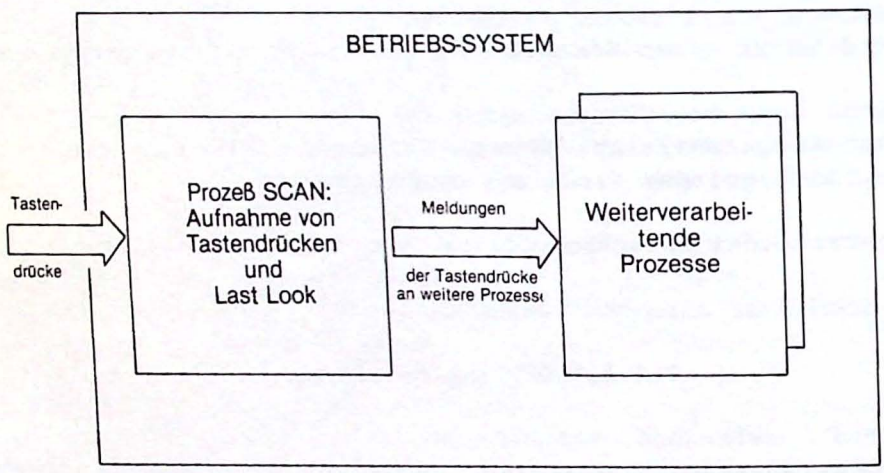

Abb. 1-6: Aufnahme und Weitergabe von Tastendrücken

Angenommen, im Prozeß SCAN ist ein Fehler im Last Look Programm, d.h. die Tastenentprellung funktioniert nicht. Falls eine Taste zweimal prellt, wird sie zweimal gemeldet, statt nur einmal. Es kommt zu einer Fehlfunktion des Gerätes.

Üblicherweise werden die verschiedenen Prozesse eines Programmsystems parallel entwickelt, d.h. die weiterverarbeitenden Prozesse stehen zum Test des Prozesses SCAN anfangs nicht zur Verfügung.

Mit Hilfe des folgenden "Offline-Testsystems" kann der Prozeß SCAN trotzdem effizient getestet werden: Statt des echten Programmssytems wird ein Testprogramm erstellt, welches anstelle der originalen weiterverarbeitenden Prozesse sogenannte "Dummy"-Prozesse enthält, die nichts weiter tun, als alle von SCAN empfangenen Meldungen auf einem Datensichtgerät auszugeben (s. Abb. 1-7).

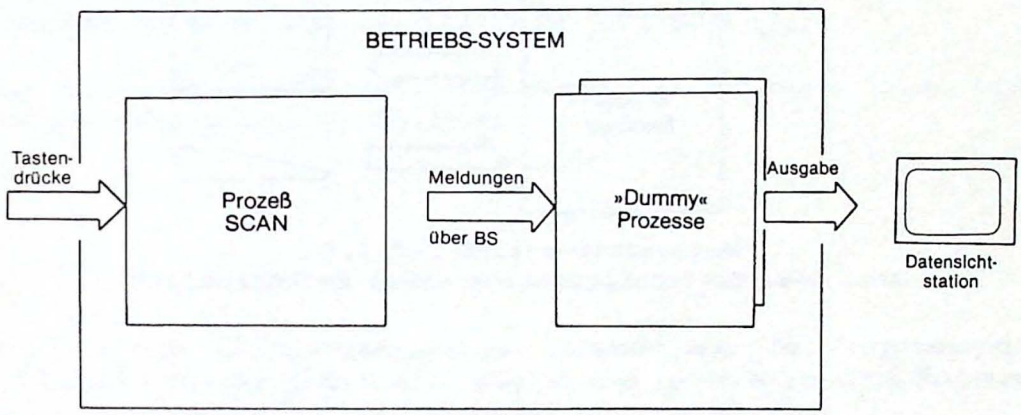

Abb. 1-7: "Dummy"-Prozesse ersetzen weiterverarbeitende Prozesse zu Testzwecken

Mit diesem einfachen Testsystem sieht der Programmierer alle Meldungen am Bildschirm, die sein Prozeß weitergibt. Die prellende Taste z.B. würde er daran erkennen, daß aufgrund nur eines Tastendruckes zwei Meldungen am Bildschirm ausgegeben werden. Der Programmierer hat so die Möglichkeit, zum Zeitpunkt der Systemintegration, d.h. zu dem Zeitpunkt, wo alle Prozesse des Systems im Zusammenspiel getestet werden sollen, einen gründlich vorgetesteten Prozeß SCAN zur Verfügung zu stellen.

Die Verallgemeinerung dieses sehr häufig auftretenden Problems, das hier etwas trivial erscheint, könnte darin bestehen, daß in einem Rechnerverbund die einzelnen Teilrechner getestet werden sollen, etwa in folgender Konfiguration (Abb. 1-8):

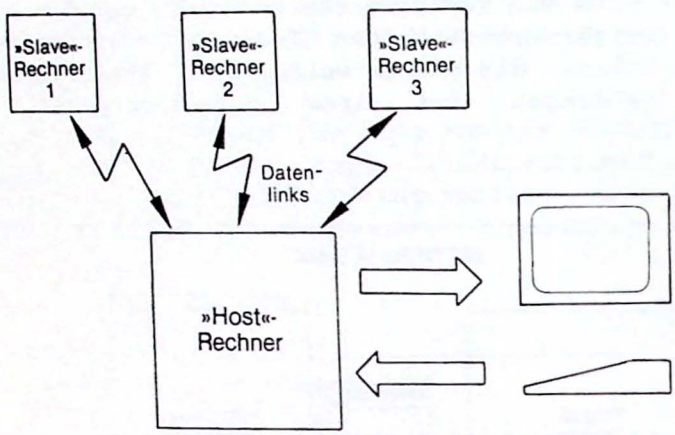

Abb. 1-8: Testkonfiguration eines Rechnernetzes

Beispiel 2

Der "Host Rechner" hat die Aufsicht über drei "Slave-Rechner". Der "Host" ist also der Herr, die "Slaves" die Diener. Die Verbindung zwischen den Rechnern könnte eine HDLC-Strecke sein (s. 1.1.4).

Für den Test der einzelnen Slaves könnte der Host wiederum so modifiziert werden, daß alle Befehle, die normalerweise aufgrund von automatischen Abläufen an die Slaves erteilt werden, per Konsole eingegeben werden. Ebenso sollen alle Rückmeldungen der Slaves am Bildschirm oder Drucker ausgegeben werden.

Auf diese Weise können die Reaktionen der Slaves ohne ein möglicherweise kompliziertes Host-Programm getestet werden.

Solche Offline-Testsysteme zu erstellen, kostet natürlich, je nach Komplexität, einen gewissen Aufwand, der bis in die Größenordnung des Aufwandes für das Originalprogramm gehen kann. Die Vorteile liegen jedoch auf der Hand:

1. Unabhängiges Testen verschiedener Prozesse während der Entwicklung.

2. Vortesten von Änderungen bei neuen Programmversionen.

3. Schnellere Fehlereingrenzung.

Offline-Testsysteme eignen sich also zum getrennten Testen einzelner Programmteile. Soll das System jedoch als Ganzes fehlerfrei arbeiten, muß das komplette Programm unter Echtzeitbedingungen getestet werden können. Es reicht dann nicht mehr aus, Schnittstellen eines Prozesses zu anderen Prozessen durch z.B Eingaben an einer Datensichtstation zu simulieren.

Man benötigt Hilfsmittel, die einen Test On-Line, also unter Realzeitbedingungen ermöglichen.

1.1.5.2 Online-Testsystem

Unter einem Online-Testsystem versteht man im Programmsystem eingebaute Testhilfsmittel, die in dem unter Echtzeitbedingungen ablaufenden vollständigen Programm zu benutzen sind.

In unseren bisherigen Beispielen sind wir davon ausgegangen, daß die Kommunikation zwischen verschiedenen Prozessen durch Meldungsaustausch (s. 2.5) stattfindet. Das Zusammenspiel verschiedener Prozesse ist dann gewährleistet, wenn zum richtigen Zeitpunkt die richtigen Meldungen von den Prozessen versendet und empfangen werden.

Man braucht ein Testvehikel, das die versendeten und empfangenen Meldungen in ihrer zeitlichen Reihenfolge in Echtzeit aufzeichnet und später bei Bedarf ausgibt.

Ein solcher "Online-Tracer", der ins Betriebssystem eingebaut ist, bietet diese Möglichkeit. Um eine hohe Effizienz zu gewährleisten, sollte er mindestens folgende Eigenschaften haben:

1. Im Realzeitbetrieb ein- und ausschaltbar, sowie auslesbar sein, z.B. vom Bedienterminal aus,

2. Genügende Speichertiefe bieten, um den Meldungsablauf hinreichend weit in die Vergangenheit verfolgen zu können (z.B. 150 Meldungen).

3. Möglichst selektiv sein, um die gewünschte Information ausfiltern zu können. Mögliche Selektionsmerkmale sind:

   Betriebssystem-Aufrufe (z.B. SEND_EVENT, RECEIVE_EVENT),

   Prozeßnummern,

   bestimmte Anwenderinformationen.

Damit könnte man z.B. alle Meldungen aufzeichnen, die der Prozeß XY mittels RECEIVE_EVENT empfängt, wobei dann noch das dritte Anwenderbyte den Wert Z haben muß.

Ein solcher Online-Tracer ist für die Software-Entwicklung ähnlich effizient, wie ein Logik-Analysator für die Hardware-Entwicklung. Die oben erwähnten Eigenschaften (Realzeitbetrieb, Speichertiefe, Selektivität) stehen im einen Fall für Software Ereignisse (Meldungen, über das Betriebssystem verschickt), im anderen Fall für Hardware Ereignisse (eingetretene Triggerbedingungen). Umso unverständlicher ist es, daß käufliche Realzeit-Betriebssysteme diese Möglichkeit selten vorsehen.

Der Einsatz von Online-Tracern zur Suche von verdeckten logischen Fehlern in komplexen Programmsystemen ist dem Gebrauch von In-Circuit-Emulatoren (ICE) wesentlich überlegen, wobei aber das Programmsystem bereits eine gewisse Mindeststabilität erreicht haben muß (z.B. keine trivialen Adressierungsfehler mehr).

Online-Testsysteme können fast beliebig erweitert werden. Die Anforderungen richten sich stark nach dem Anwendungsfall. Sehr nützliche, allgemein verwendbare Funktionen sind z.B. Monitorfunktionen, mit denen Speicherbereiche im Betrieb kontrolliert und bei Bedarf geändert werden können. Weitere nützliche Eigenschaften sind Warteschlangenfunktionen, mit denen die Inhalte von Warteschlangen angeschaut werden können und evtl. neue Elemente eingebracht werden können. Für die notwendigen Ein- und Ausgaben muß natürlich ein geeignetes Ein-/Ausgabesystem (s. 1.1.2) zur Verfügung stehen.

Die folgende Abbildung faßt noch einmal die in diesem Kapitel erwähnten Teilkomplexe eines Betriebssystems zusammen (Abb. 1-9):

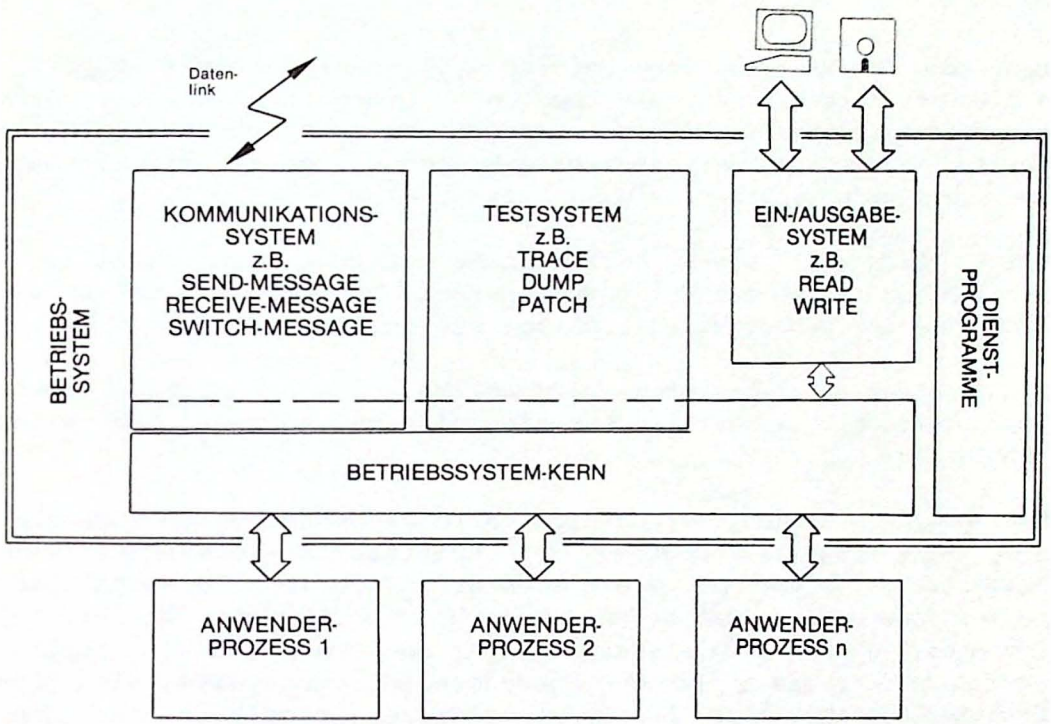

Abb. 1-9: Komponenten eines Betriebssystems

Die Einbettung von Kommunikationssystem und Testsystem in den Betriebssystem-Kern soll veranschaulichen, daß zur Realisierung dieser Funktionen Eingriffe in den Betriebssystem-Kern notwendig sind, während das Ein-/Ausgabesystem wie ein Anwenderprozeß erstellt werden kann. Die Dienstprogramme sind als getrennter Block dargestellt, da sie meistens, losgelöst vom Realzeitbetrieb eines Programmsystems, wie Anwenderprozesse benutzt werden.

Damit ist die Behandlung derjenigen Funktionen eines Betriebssystems, die über den Betriebssystem-Kern hinausgehen, abgeschlossen. Es sollten einige Beispiele gegeben werden, ohne vollständig alle Möglichkeiten, die ein Betriebssystem bieten könnte, abzuhandeln.

1.2 Ein-Benutzer, Mehr-Benutzer-Systeme

Mehr-Benutzer(Multitasking)-Betriebssysteme sind solche, die verschiedene Aufgaben zur gleichen Zeit bearbeiten können. 'Zur gleichen Zeit' heißt, daß die Umschaltung zwischen den einzelnen Aufgaben durch das Betriebssystem so schnell geht, daß sie nach außen hin nicht spürbar ist.

Unter 'Benutzer' einer Rechenanlage versteht man nicht ausschließlich einen menschlichen Benutzer, sondern allgemein ein Programm, das gerade auf der Anlage bearbeitet wird.

Ein-Benutzer-Betriebssysteme sind solche, die auch über längere Zeit betrachtet, jeweils nur eine einzige Aufgabe bearbeiten können.

Man kann sich fragen, warum nicht alle Rechenanlagen, insbesondere auch Personal Computer, mit Multitasking-Betriebssystemen ausgestattet sind, da diese doch offensichtlich die Maschine besser ausnutzen. Die bekanntesten Betriebssysteme für Personal Computer, wie das Betriebssystem für den IBM Personal Computer oder CP/M oder das Apple oder Commodore Betriebssystem, sind Ein-Benutzer-Systeme. Zur Beantwortung dieser Frage kann man drei Gründe anführen:

1. Komplexität

Es ist heute noch so, daß die meisten Personal Computer die für einen wirkungsvollen Mehr-Benutzer Betrieb notwendige Komplexität der Hardware nicht bieten, insbesondere oft keine genügend schnellen Massenspeicher zur Verfügung stehen. Bei den meisten Aufgaben für Personal Computer muß oft auf Massenspeicher, z.B. Disketten zugegriffen werden. Sollen sich nun etwa 5 Anwender eine einzige Diskettenstation teilen, wird die Reaktionszeit des Systems im wesentlichen durch die Zugriffszeiten auf die Disketten bestimmt. Außerdem müßten alle Benutzer ihre Programme auf derselben Diskette abgelegt haben. Noch absurder wird die Situation, wenn nur ein Kassettenrecorder als Speichermedium verfügbar ist.

Man muß jedoch feststellen, daß der Trend zu schnelleren und auch billigeren Massenspeichern vorhanden ist. Winchester-Plattenlaufwerke und Tintenstrahldrucker sind Beispiele.

Schreib-Lese-Speicher (RAM-Speicher) ist heute schon so billig und kompakt, daß neuere Personal Computer oft mehr als 128 kByte RAM-Speicher bieten.

2. Kompatibilität

Der Gesamtpreis einer Rechenanlage wird in Zukunft mehr und mehr von der Software bestimmt werden. Die folgende Grafik (Abb: 1-10) zeigt für 1983 das Verhältnis von Hardware- zu Softwarekosten eines Personal Computers (Osborne), der mit verschiedenen gängigen Softwareprodukten ausgestattet wurde.

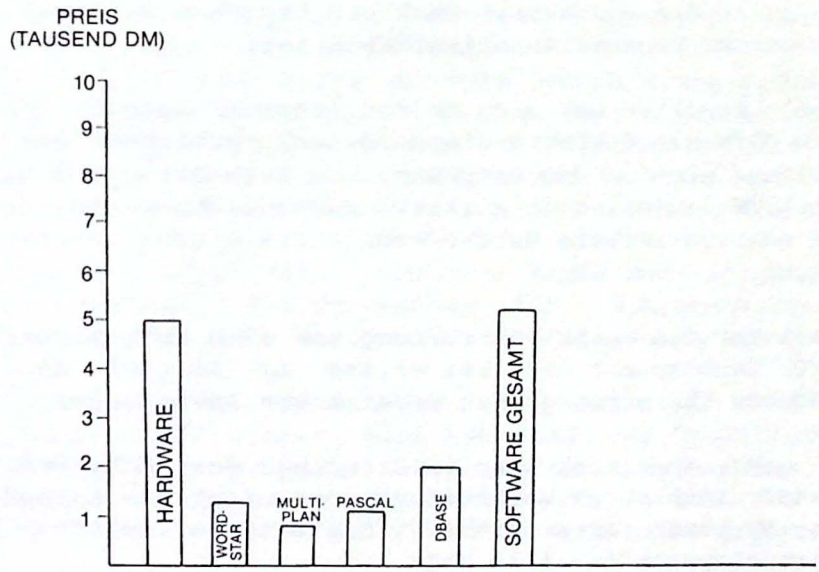

Abb. 1-10: Kostenvergleich Hardware - Software

Ein Anwender verlangt deshalb, daß er seine teuren Softwareprodukte weiterverwenden kann, wenn er eine neue Hardware kauft. Diese Forderung ist für einen Hersteller natürlich wesentlich leichter zu erfüllen, wenn er auf einer weiterentwickelten Hardware dasselbe Betriebssystem benutzt.

Dieser Punkt ist eng verknüpft mit dem dritten Punkt:

3. Standardisierung

Für die Hardware von Rechenanlagen gibt es heute vielfältige Standards. Bekannte Hardware Standards sind z.B. die V.24 Schnittstelle für asynchronen, seriellen Datenverkehr, die sog. IEC-Bus Schnittstelle für parallele Datenübertragung, die Centronics Druckerschnittstelle, die X.21 Schnittstelle usw.

Das bedeutet, daß man z.B. Drucker eines Herstellers, der sich an diese Norm hält, an einer Vielzahl von Rechnern anschließen kann, die ebenfalls diese Norm erfüllen.

Bei Software-Produkten sieht die Situation leider ganz anders aus. Hier ist es die Ausnahme, wenn ein Programm auf zwei Rechnern verschiedener Hersteller ablauffähig ist.

Für Personal Computer hat sich in den letzten Jahren das Betriebssystem CP/M als de-facto Standard herausgebildet. Das heißt nun noch lange nicht, daß Programme von Rechner A und B, die beide unter CP/M laufen, auch wirklich austauschbar sind, da z.B. die in CP/M konfigurierbare Hardwareansteuerung ganz unterschiedlich sein kann.

Einen Vorteil hat die weite Verbreitung von CP/M aber sicherlich: Es gibt für Rechner mit CP/M bei weitem am meisten käufliche Softwareprodukte für einen großen Bereich von Anwendungen.

Der Trend geht aber auch hier in Richtung der Mehr-Benutzer-Systeme, wenn auch deren Verbreitung noch nicht das Ausmaß von Ein-Benutzer-Systemen erreicht hat. Ein Beispiel ist das Timesharing-Betriebssystem (s. 1.3) UNIX.

Die oben gestellte Frage, warum nicht mehr Personal Computer mit Mehr-Benutzer-Betriebssystemen ausgestattet sind, kann auch anders gestellt werden:

Für welche Anwendungen sind heute Mehr-Benutzer-Systeme sinnvoll einzusetzen?

Für Mikrocomputer gibt es heute zwei Hauptanwendungen:

1. Universell einsetzbare Systeme.
2. Speziell einsetzbare Systeme.

Zum ersten Bereich rechnen wir z.B. Personal Computer. Diese sind nicht von vorneherein auf bestimmte Aufgaben ausgelegt, sondern können, je nach geladenem Programm, z.B. Textverarbeitung oder Finanzkalkulation durchführen. Die Hersteller solcher Systeme haben beim Umstieg auf neue Betriebssysteme mit den drei erläuterten Punkten

- Komplexität,
- Kompatibilität,
- Standardisierung

zu kämpfen.

Zum zweiten Bereich gehören Industriesteuerungen. Diese sind für einen ganz speziellen Anwendungszweck entworfen. Hier kann der Anwender meist auch nicht Hardware und Software getrennt kaufen oder gar Fremdsoftware auf der Anlage laufen lassen, sondern er kauft das System als Einheit aus Hardware und Software. Hier sind zumindest die Punkte Kompatibilität und Standardisierung keine so zwingende Notwendigkeit wie bei universell einsetzbaren Rechnern.

Das mag der Grund sein, warum es heute noch so gut wie keine standardisierten Softwarepakete für Mikrocomputer-Industriesteuerungen gibt, die herstellerunabhängig einsetzbar wären.

Andererseits haben Entwickler und Hersteller von Industriesteuerungen die Freiheit, eine dem Stand der Technik entsprechende Software zu entwickeln. Da viele der Aufgaben für Industriesteuerungen in Echtzeit ablaufen müssen, bietet sich für diese Probleme der Einsatz eines Realzeit Multitasking Betriebssystems an.

Hier gibt es für bestimmte Hardware die Betriebssysteme zu kaufen, z.B. das Betriebssystem RMOS (s. 4.2) für das SMP-Europakartensystem (Siemens)

1.3 Realzeit-Fähigkeit

Realzeitfähigkeit eines Systems bedeutet, daß es innerhalb einer vorgegebenen Maximalzeit aufgrund abgestimmter äußerer Einwirkungen auf das System bestimmte Aktionen vornehmen muß. Für viele Anwendungen ist Realzeitverhalten nicht erforderlich. Beim Plotten von Kurven (d.h. Kurven von einem speziellen Zeichengerät durch den Rechner zeichnen lassen) ist es egal, ob die Aktion 10 Sekunden länger dauert oder nicht. Beim Ausdrucken von Listen ist ebenfalls Echtzeitverhalten nicht erforderlich.

Ein bekanntes Beispiel für Multitasking-Betrieb, das kein Realzeitverhalten erfordert, ist der Timesharingbetrieb (Zeitscheibenbetrieb, s. 2.6.4.2) eines Rechenzentrums. Hier bekommt ein Anwender eine bestimmte Rechenzeit zugeteilt, bevor der nächste Anwender an der Reihe ist, beispielsweise 100 msec. Je mehr Anwender teilnehmen, desto länger wird die Wartezeit für den Einzelnen.

Das bekannteste Timesharing-Betriebssystem für Mikrocomputer ist UNIX von den Bell-Laboratories.

In den folgenden Beispielen ist Realzeitfähigkeit notwendig:

1. Der im einführenden Beispiel erwähnte Telefoncomputer muß als Realzeitsystem entworfen sein, da der Benutzer auf Tastendruck sofortige (d.h. nach maximal 1/10 sec) Reaktion erwartet.

2. Ein Schweißroboter muß in Realzeit arbeiten, da er in der Zeit, in der das Fließband an ihm vorbeifährt, seine Schweißpunkte gesetzt haben muß.

Typische Anwendungen für Realzeitsysteme sind Steuerungen industrieller Prozesse. Ein wichtiges Merkmal von Realzeitsystemen ist, daß sie unterbrechungsgesteuert arbeiten. Das heißt, daß eine vom Rechner gerade durchgeführte Aktion jederzeit unterbrochen werden kann, um eine wichtigere Tätigkeit vorzunehmen (s. 5.1.4, 6.2.4.1).

Das bedeutet auch, daß wichtige Programmteile resident im Speicher vorhanden sein müssen, wo sie in einem Nur-Lese-Speicher (ROM) abgelegt sein können, oder einmal beim Hochlauf der Anlage in einen RAM-Speicher geladen werden.

Das Laden solcher kritischer Programmteile von der Platte erst zur Ausführungszeit, wäre für viele Probleme zu zeitaufwendig, um noch innerhalb der geforderten Reaktionszeit zu bleiben.

In folgender Tabelle sind nochmals einige typische Eigenschaften eines Realzeit-Systems zusammengestellt.

Tabelle 1-1: Notwendige Eigenschaften eines Realzeitsystems

Maximale Reaktionszeit auf äußere Ereignisse	10-100 Millisek.
Umschaltung zw. verschiedenen Aufgaben	Interruptgesteuert
Wichtige Programmteile	Nach Anlagenhochlauf resident im Speicher

Man kann jedoch nicht allgemein sagen, daß von der Effizienz der Auslastung der Rechenanlage her ein Realzeit-Betriebssystem einem Timesharing-Betriebssystem vorzuziehen wäre, wie das beim Vergleich Ein-Benutzer-Systeme, Mehr-Benutzer-Systeme zutraf. Der Grund ist der, daß die Kriterien, nach denen die Rechenanlage dem einzelnen 'Benutzer' zugeteilt wird, verschieden sind.

Eine gängige Situation in einem Timesharing System ist die, daß ein Anwender gerade mit einem Texteditor arbeitet, während ein zweiter Benutzer einen Compilerlauf durchführen läßt. Das Betriebssystem muß beiden den Zugang zum Rechner gewähren, insbesondere muß also der Compilerlauf von Zeit zu Zeit unterbrochen werden, um dem anderen Benutzer Eingaben an der Konsole zu erlauben.

Die gleiche Situation in einem typischen Realzeitsystem hätte ein ganz anderes Ergebnis:

Nach einmal gestartetem Compilerlauf müßte der Benutzer des Texteditors für eine Eingabe an der Konsole das Ende des Compiler-

laufes abwarten, falls beide gleiche Priorität hätten. Es besteht keine Notwendigkeit zum Prozeßwechsel, da

- in Realzeitsystemen die Laufzeiten von Prozessen normalerweise nicht so hoch sind, daß gleichberechtigte Wartende dadurch blockiert würden.

- in Realzeitsystemen wichtigere Programmteile dank höherer Priorität die Unwichtigeren sofort verdrängen können.

- jeder Prozeßwechsel die Effizienz des Systems belastet und daher zu vermeiden ist, wenn er umgangen werden kann.

Arbeitet man mit einem Realzeitsystem ist es für den Programmierer von großer Wichtigkeit, den Mechanismus zu kennen, nach dem die Prozesse umgeschaltet werden (den sog. 'Scheduling-Algorithmus', s. 2.6.4.2). Dadurch wird festgelgt, wann z.B. Programmteile nicht unterbrochen werden können bzw. wann sie unterbrochen werden können und der zugehörige Prozeß verdrängt werden darf. Auf diese Problematik wird noch eingegangen werden (s. 6.2.3).

Der Schlüssel zum Realzeitverhalten eines Systems ist der Betriebssystem-Kern. Deshalb wird in den folgenden Kapiteln ausführlich auf den Aufbau und die Eigenschaften heute gängiger Betriebssystem-Kerne eingegangen.

2. Betriebssystem-Kern

2.1 Prozesse

Prozesse nennt man die Programme, die vom Betriebssystem-Kern verwaltet werden. Der Begriff 'Prozeß' wird in den meisten gängigen Programmiersprachen nicht verwendet. Höhere Programmiersprachen verwenden die Begriffe 'Modul' und 'Prozedur'. Sie haben eine genau definierte Bedeutung.

Als Modul bezeichnet man ein zusammenhängend übersetzbares (kompilierbares oder assemblierbares) Programmstück. Ein Programmsystem kann damit in kleinere Stücke unterteilt werden, man spricht auch von 'Modularer Programmierung'. Der Vorteil liegt darin, daß bei der Entwicklung die Programmteile (Moduln) von verschiedenen Entwicklern bearbeitet werden können, die die übrigen Moduln des Systems nicht oder nur an den Schnittstellen zu ihrem Modul kennen müssen. Auch muß bei der Behebung von Programmfehlern nicht das ganze Programmsystem neu kompiliert oder assembliert werden, es reicht oftmals, den fehlerhaften Modul neu zu übersetzen.

Werden gleiche Programmteile an mehreren Stellen des Programms benötigt, so faßt man sie in Prozeduren (Unterprogrammen) zusammen.

Für die Anwendung eines Betriebssytems reichen diese beiden Begriffe nicht mehr aus. Das Prozeßkonzept bietet hier erweiterte Möglichkeiten (Kapitel 6). Im Gegensatz zu Modul und Prozedur ist der Prozeß ein abstrakter Begriff, an den keine besonderen formalen Forderungen gestellt werden. Ein Prozeß kann deshalb aus einem oder aus mehreren Moduln bestehen, er muß keine, kann jedoch sehr viele Prozeduren enthalten. Der Begriff 'Prozeß' wird immer dann verwendet, wenn Programmteile (Moduln, Prozeduren) zu einer Programmeinheit zusammengefaßt werden und diese von einem Betriebssystem verwaltet wird.

Die Aufgabe eines Multitasking-Betriebssystems ist es, für den ordnungsgemäßen Ablauf derjenigen Programmteile zu sorgen, die als Prozeß realisiert sind. Ein Multitasking-Betriebssystem hat nur dann Sinn, wenn der Ablauf von mindestens 2 Prozessen zu überwachen ist.

Betriebssysteme für Mikrocomputer sind je nach Anwendungsfall für unter 10 bis über 2000 Prozesse ausgelegt. Z.B. kann das Betriebssystem ROSY (Realtime Operating System) 10 Prozesse überwachen, während das Betriebssystem MTOS (käufliches Betriebssystem der Fa. ipi, vgl. Kapitel 4.2) bis zu 2048 Prozesse verwaltet. (Der Source-Code des Betriebssystems ROSY ist im Anhang dieses Buches abgedruckt, in Kapitel 3.4 ist es näher beschrieben). Das Betriebssystem teilt den Prozessen nach genau festgelegten Regeln (Kapitel 2.6.4) den Prozessor zu.

Hat ein Prozeß den Prozessor zugeteilt bekommen, arbeitet dieser solange das Programm dieses Prozesses ab, bis das Betriebssystem entscheidet, daß ein anderer Prozeß den Prozessor zugeteilt erhält. Man spricht hier auch von einer 'virtuellen Maschine'. Hardware und Betriebssystem arbeiten für jeden Prozeß wie eine virtuelle Maschine, d.h. für jeden Prozeß erscheinen Hardware und Betriebssystem so, als hätte er diese für sich allein.

Da der Rechner immer nur sequentiell arbeiten kann (Multiprozessorsysteme seien der Einfachheit halber von der Betrachtung ausgeschlossen), muß er also von Zeit zu Zeit die Bearbeitung der Programme, die zu einem Prozeß gehören, verlassen und muß auf die Programme des Betriebssystems umschalten, um zu entscheiden, ob ein anderer Prozeß gestartet werden muß. Ist dies der Fall, so wird der gerade laufende Prozeß 'zur Ruhe gesetzt', d.h. sämtliche Informationen über das gerade bearbeitete Programm werden gespeichert, damit es später an dieser Stelle ordnungsgemäß fortgesetzt werden kann. Diesen Entscheidungsvorgang durch das Betriebssystem nennt man 'Scheduling', den die Entscheidung treffenden Programmteil 'Scheduler'. Näher wird im Kapitel 2.6.4.2 auf das Scheduling eingegangen.

Damit ist auch der Begriff 'Multitasking' etwas näher erklärt. Der Scheduler wird nämlich manchmal auch als 'Task Multiplexer' (Aufgabenumschalter) bezeichnet. Schematisch läßt sich die Aktion des Betriebssystems in Analogie zu einem Hardware-Multiplexer darstellen (Abb. 2-1).

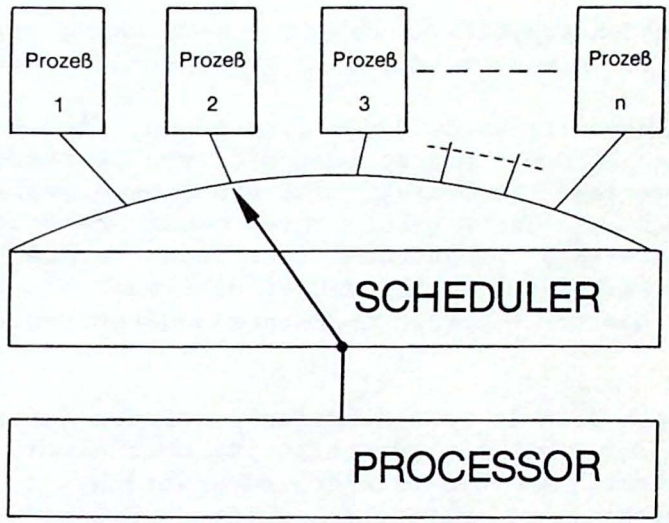

Abb. 2-1: Scheduler

Die Zeit, wie schnell der Umschalter springt, sowie die Methode nach der der Schalter gedreht wird, ist unterschiedlich. In den Kapiteln 2.6 wird näher darauf eingegangen. Dort wird auch erklärt, welche Parameter einen Prozeß für das Betriebssystem als solchen kennzeichnen.

Der Realzeitbezug bleibt für die Prozesse natürlich nur erhalten, wenn die einzelnen Prozesse keine zu langen Laufzeiten haben bis wieder der Scheduler zum Ablauf kommt. Um eine Überlast zu vermeiden, stehen Methoden zur Abwehr dieser zur Verfügung (Kapitel 6.2.3).

2.1.1 Prozeßzustände

In den folgenden Kapiteln werden im Zusammenhang mit Prozessen einige Begriffe verwendet, die hier vorab erklärt werden sollen.

Im vorigen Abschnitt wurde davon gesprochen, daß Prozesse 'zur Ruhe gesetzt' werden, man spricht hier von 'vertagen' und 'verdrängen'. 'Vertagen' bedeutet, daß ein Prozeß selbständig entscheidet, daß er den Prozessor nicht mehr benötigt, bis ein bestimmtes Ereignis eingetreten ist. Von 'Verdrängen' spricht man, wenn einem Prozeß, ohne daß er dies wünscht, vom Betriebssystem der Prozessor entzogen und einem anderen Prozeß zugeteilt wird.

Ein Prozeß kann sich in verschiedenen Zuständen befinden, je nach dem ob ihm der Prozessor zugeteilt ist oder nicht. Hat er den Prozessor nicht, so muß unterschieden werden, ob er auf den Prozessor wartet (z.B. der Prozeß wurde verdrängt) oder ob dies nicht der Fall ist (z.B. der Prozeß hat sich selbständig vertagt).

Die wichtigsten Prozeßzustände und ihre Bedeutung im Einzelnen:

- 'Ruhend': Der Prozeß ist neu erzeugt aber noch nicht gestartet worden.
- 'Running': Dem Prozeß ist der Prozessor zugeteilt.
- 'Ablaufbereit': Der Prozeß wartet auf die Zuteilung des Prozessors.
- 'Warten auf Ereignis': Der Prozeß ist vertagt und wartet auf ein bestimmtes Ereignis. Die möglichen Ereignisse sind im Betriebssystem festgelegt. Für jedes Ereignis wird ein eigener Wartezustand geführt.

Abb. 2-2 zeigt die möglichen Zustandsübergänge eines Prozesses.

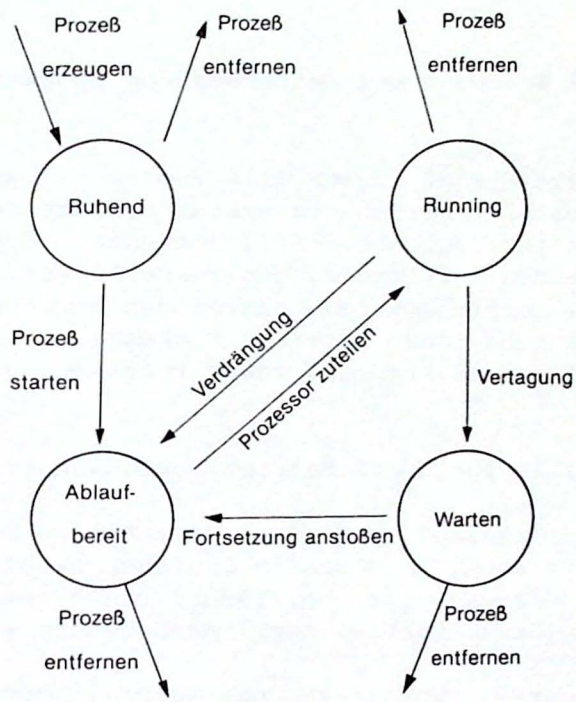

Abb. 2-2: Zustandsübergänge eines Prozesses

Ein im System noch nicht vorhandener Prozeß kann mit einem besonderen Betriebssystemaufruf (z.B. CREATE_PROCESS) dem System bekannt gemacht werden. Er ist dann im Zustand 'Ruhe' und kann mit einem Betriebssystemaufruf (z.B. START_PROCESS) in den Zustand 'Ablaufbereit' geführt werden. Das Entfernen eines Prozesses aus dem System ist aus allen Zuständen möglich.

Ist der Prozeß im Zustand 'Warten auf Ereignis', so wird mit dem Auftreten dieses Ereignisses der Prozeß ablaufbereit gemacht, d.h., das Ereignis stößt seine Fortsetzung an. Einem ablaufbereiten Prozeß wird vom Scheduler der Prozessor zugeteilt, der Prozeß befindet sich nach der Zuteilung im Zustand 'Running'. Diesen Zustand verläßt er entweder indem er sich vertagt (neuer Zustand: 'Warten auf Ereignis') oder indem er verdrängt wird, d.h. er ist

dann weiterhin ablaufbereit und wartet auf die Zuteilung des Prozessors.

2.1.2 Erzeugen und Entfernen von Prozessen

In einem Realzeit-System müssen alle Prozesse, an die Realzeitanforderungen gestellt werden, im System präsent sein, so daß sie jederzeit sofort ihre Aufgabe erfüllen können. Für Prozesse, an die keine zwingenden Zeitforderungen gestellt werden, trifft dies nicht zu. Viele Betriebssysteme bieten dem Anwender deshalb die Möglichkeit, während des Betriebs Prozesse neu in das System einzubringen und sie zu starten, sowie Prozesse aus dem System zu entfernen.

Anwendungsbeispiele für diese Betriebssystem-Funktionen sind:

- Nachladbare Prozesse: Prozesse, die selten benötigt werden, können in Systemen, in denen im Speicher nicht Platz für alle vorhandenen Prozesse ist, bei Bedarf von einem Speichermedium (z.B. Floppy-Disk, Bubble, etc.) nachgeladen werden.

- Prozesse, deren Funktionen nur einmal oder sehr selten benötigt werden. Ein Beispiel hiefür ist ein sog. Hochlaufprozeß, der beim Starten des Systems die Initialisierung von Teilen des Betriebssystems sowie das Starten der Prozesse übernimmt. Danach kann er aus dem System entfernt werden. Auch ein Prozeß, der auf Anforderung eine Statistik erstellt, ist ein Beispiel für einen Prozeß, der nicht laufend vorhanden sein muß.

Für das Erzeugen und Entfernen von Prozessen stehen folgende Funktionsaufrufe des Betriebssystems zur Verfügung:

- CREATE_PROCESS: Der Prozeß wird neu in die Verwaltung des Betriebssystems aufgenommen. Die benötigten Speicherbereiche (z.B. Stack) werden ihm zugewiesen. Damit ist der Prozeß aber noch nicht ablaufbereit.
- START_PROCESS: Das Betriebssystem übernimmt den Prozeß in seine PROCESS_QUEUEs und bringt ihn damit in den Zustand 'Ablaufbereit'. Er kann jetzt vom PROCESS_SCHEDULER gestartet werden.

- DELETE_PROCESS: Der Prozeß wird aus der Prozeßverwaltung des Betriebssystems entfernt. War dieser Prozeß ein nachladbarer Prozeß, so ist dessen Speicherbereich jetzt wieder frei und kann vom Betriebssystem erneut vergeben werden.

Der Anstoß zum Erzeugen und Starten eines Prozesses kommt durch den entsprechenden Betriebssystemaufruf von einem anderen Prozeß. Beim Entfernen eines Prozesses kann der Anreiz entweder von dem Prozeß selbst oder auch von einem anderen Prozeß kommen.

2.2 Warteschlangen

Im vorherigen Abschnitt haben wir eines der Grundprinzipien von Multitasking-Betriebssystemen, das Prozeßkonzept, kennengelernt. Hier soll nun ein zweites vorgestellt werden, die 'Warteschlange' oder 'Queue', wobei an dieser Stelle nur der Grundgedanke vermittelt werden soll. Auf die Anwendung wird in den nächsten Kapiteln ausführlich eingegangen.

Warteschlangen sind ein Hilfsmittel zur Zwischenspeicherung von Daten. In einem Multitasking-Betriebssystem werden Warteschlangen an vielen Stellen benötigt, z.B.

- für Prozesse, die nicht vertagt sind (PROCESS_QUEUEs, Kapitel 2.6.1).
- für Mitteilungen an einen anderen Prozeß, die auf die Bearbeitung durch diesen Prozeß warten (EVENT_QUEUEs, Kapitel 2.5.1).
- für Zeitaufträge, die das Betriebssystem für seine Prozesse verwaltet (TIMER-QUEUES, TIME-SLOT-TABLE, Kapitel 2.7).

Über das Betriebssystem hinaus sind Warteschlangen, die vom Anwender allein verwaltet werden, ebenfalls oft sinnvoll ('Vertagen von Meldungen', Kapitel 7.3.5). In diesem Fall stellt das Betriebssystem nur den Warteschlangen-Mechanismus zur Verfügung, die Daten in der Warteschlange interessieren das Betriebssystem dann nicht.

Allgemein läßt sich sagen, daß Warteschlangen ein wichtiges Arbeitsprinzip für Multitasking-Betriebssysteme sind. Daher wird im

folgenden Abschnitt ausführlich auf verschiedene Realisierungsmöglichkeiten und Anwendungen eingegangen.

Verdeutlichen wir uns das Prinzip der Warteschlange an einem alltäglichen Beispiel: In einem Selbstbedienungsladen sind die zur Verfügung stehenden Einkaufswagen ineinandergeschoben in einer Reihe (Warteschlange) aufgereiht. Jeder Kunde nimmt sich beim Betreten des Geschäfts einen Wagen, vor dem Verlassen stellt er ihn wieder in die Schlange zurück. Dabei stehen 2 Möglichkeiten zur Verfügung:

2.2.1 FIFO-Prinzip

Bei einem Supermarkt mit Drehkreuz am Eingang sind oftmals die Einkaufswagen unter der Absperrung durchgeschoben. Der Kunde geht durch das Drehkreuz und nimmt sich vom vorderen Ende der Wagenreihe einen weg. Hat er seinen Einkauf beendet und den Wagen geleert (die Waren in Taschen oder im Auto verstaut), so schiebt er den leeren Wagen wieder hinten in die Einkaufswagenreihe. Mit jedem Wagen, der vorne weggenommen wird, wandert dieser Wagen in der Warteschlange um eine Position nach vorne, bis er wieder der vorderste ist und vom nächsten Kunden genommen wird. Wir erkennen dabei, daß die Wagen in der Reihenfolge, in der sie in die Schlange eingefügt wurden, auch wieder verwendet werden und daß der am längsten in der Schlange stehende Wagen auch als erster genommen wird.

Dieses Prinzip wird als FIFO-Prinzip (First In -First Out) bezeichnet. Aus der Digitaltechnik ist es manchem Leser sicherlich ein Begriff. Es handelt sich dort um einen Hardware-Speicherbaustein, bei dem in das Eingabeport Information eingeschrieben wird, während sie von einem anderen Port, dem Ausgabeport, später wieder gelesen wird. Die Reihenfolge der eintreffenden Informationen bleibt dabei erhalten. Abb. 2-3 zeigt ein vereinfachtes Blockschaltbild eines Hardware-FIFO's.

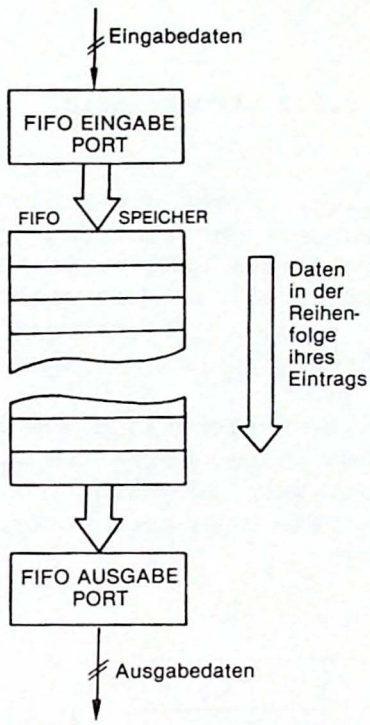

Abb. 2-3: FIFO-Pronzip

Auf die Prozesse eines Multitasking-Betriebssystems angewandt bedeutet dies folgendes:

Würde man alle Prozesse, die auf ihren Ablauf warten, in einem FIFO aufbewahren, so müßte der Scheduler, wenn er sich für einen neuen Prozeß entscheidet, nur das FIFO auslesen. Er erhielte damit automatisch den am längsten wartenden Prozeß. Tatsächlich werden die wartenden Prozesse in einer Warteschlange aufgesammelt. Man sagt auch, die Prozesse werden in eine Warteschlange 'eingekettet'.

Für das Starten der Prozesse vom Scheduler ist die Wartezeit allerdings nur eines von mehreren Kriterien. Auf die weiteren Kriterien wird im Kapitel 2.6.4.2 eingegangen. Auch wird in den

folgenden Kapiteln die softwaremäßige Realisierung der Warteschlange näher erläutert.

2.2.2 LIFO-Prinzip

Zurück zu unserem Beispiel: Steht die Schlange der Einkaufswagen nicht frei im Raum, sondern das eine Ende gegen eine Wand, so wird der Kunde sich vom freien Ende einen Wagen nehmen und ihn auch wieder dorthin zurückstellen. Man sieht, daß der zuletzt in die Schlange geschobene Wagen als erstes wieder genommen wird. Man nennt dieses Verfahren LIFO-Prinzip (Last In - First Out).

Das LIFO-Prinzip ist vielen sicherlich vom Stackmechanismus ein Begriff. Abb. 2-4 zeigt diesen Mechanismus. Erfolgt ein neuer Eintrag in den Stack, so wird die nächste freie Stackzelle verwendet. Wird ein Datum aus dem Stack entfernt, so ist es immer das zuletzt eingetragene.

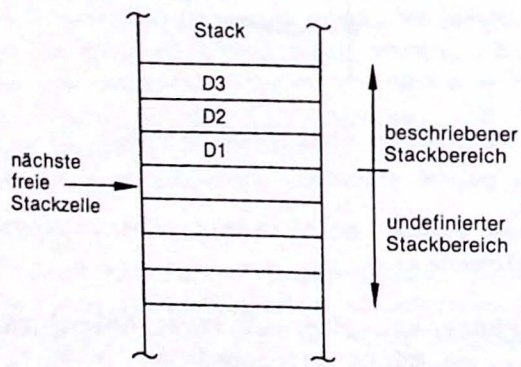

Abb. 2-4: LIFO-Prinzip

Auf die Prozesse unseres Multitasking-Betriebssystem bezogen, ergäbe dies eine Möglichkeit, Prozesse zu priorisieren. Man müßte die Prozesse an den Anfang der Prozeßwarteschlange setzen und der Scheduler würde sie vor den ans Ende eingehängten Prozessen zum Ablauf bringen. In der Praxis wird dieses Verfahren zur Prozeßpriorisierung nicht eingesetzt, es stehen dafür wesentlich

bessere Methoden zur Verfügung (Kapitel 2.6.1). Im Vorgriff auf folgende Kapitel soll hier nur gesagt werden, daß das LIFO-Prinzip zum Priorisieren bestimmter Meldungen (Kapitel 2.5.1), angewandt wird.

In den folgenden beiden Kapiteln soll nun dargestellt werden, welche Mittel ein Multitasking-Betriebssystem dem Anwender, d.h. den Prozessen, zur Verfügung stellt, um ihre Anforderungen an Speicherplatz, Datenschutz, Realzeitverhalten und Kommunikation zu befriedigen.

Es werden die wichtigsten allgemeinen Prinzipien von Realtime-Betriebssystemen vorgestellt. Die Ausführungen sind als beispielhaft anzusehen. Von den vielen verschiedenen Realisierungsmöglichkeiten werden nicht alle angesprochen. Die beschriebenen Betriebsmittel sind aber sowohl für kleine als auch für größere Betriebssysteme typisch. Nach dem Studium dieses Buches sollte der Leser in der Lage sein, die Vorgehensweise von Betriebssystemen mit Funktionsabläufen, die sich von den hier beschriebenen unterscheiden, zu verstehen.

2.3 Speicher-Pool-Verwaltung

Wie wir bereits erfahren haben, ist es immer dann sinnvoll ein Multitask-Betriebssystem einzusetzen, wenn mehrere voneinander unabhängige Aufgaben quasi gleichzeitig auf einem Prozessor in Echtzeitbetrieb bewältigt werden sollen. Diese einzelnen Aufgaben sind in Prozessen realisiert. Die Prozesse sind in Realzeitsystemen meistens nicht vollständig unabhängig voneinander, sondern haben gegenseitige Schnittstellen.

Das Zusammenspiel aller Prozesse realisiert die Gesamtfunktion des Rechnersystems. Zusammenspiel bedeutet nicht, daß die Prozesse der Reihe nach vom Betriebssystem gestartet werden und wenn der letzte beendet wurde, wieder mit dem ersten begonnen wird. Das reicht in den meisten Anwendungsfällen nicht aus.

Vielmehr ist es so, daß jeder Prozeß eine bestimmte Aufgabe zu erledigen hat. Damit er diese Aufgabe erledigen kann, müssen erst einmal die Voraussetzungen zur Erfüllung der Aufgaben gegeben

sein. Erst dann, und nur dann, darf das Betriebssystem den Prozeß zum Ablauf bringen. Daß die Voraussetzungen vorliegen, muß dem Betriebssystem in geeigneter Weise mitgeteilt werden (Kapitel 2.5). Hierzu folgendes Beispiel:

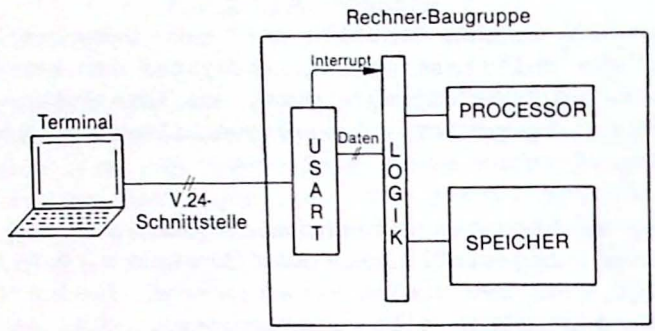

Abb. 2-5: Hardware-Komponenten

An einer Rechner-Hardware ist über eine V.24-Schnittstelle mit niederer Baudrate (<=9600) ein Terminal angesteckt (Abb. 2-5). Auf der Baugruppe läuft ein Multitasking-Betriebssystem mit mehreren Prozessen (Abb. 2-6). Ein Prozeß, er sei 'AUSGABE-PROZESS'genannt, hat die Aufgabe, die Ausgabe der Zeichen an die V.24-Schnittstelle (und damit an das Terminal) zu steuern. Die Bedienung der V.24-Schnittstelle erfolgt auf der Baugruppe durch einen USART (LSI-Bauelement,UniversalSynchronous/Asynchronous Receiver/Transmitter, z.B. Intel 8251).

Der AUSGABE-PROZESS hat dem USART die auszugebenden Zeichen einzeln zu übergeben, d.h. die Übergabe des nächsten Zeichens darf erst erfolgen, wenn der USART mit der Übertragung des vorherigen Zeichens begonnen hat. Er teilt dies durch einen Interrupt mit. Das Betriebssystem hat also bei jedem erneuten USART-Interrupt den AUSGABE-PROZESS zu starten.

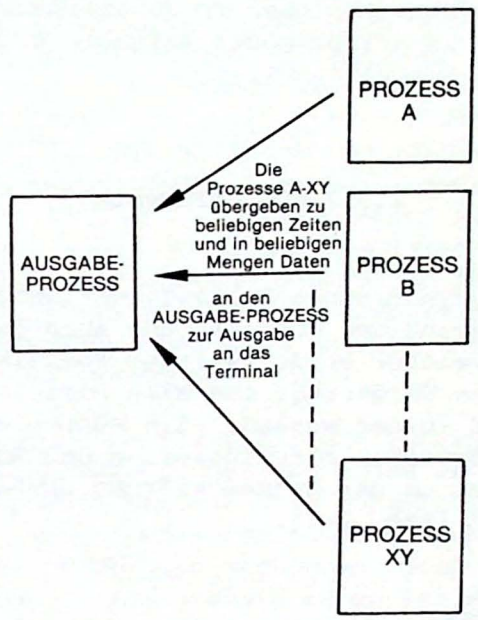

Abb. 2-6: Informationsübergabe an AUSGABE-PROZESSOR

Die anderen Prozesse veranlassen in bestimmten Fällen eine Ausgabe an das Terminal. Die genauen Aufgaben dieser Prozesse sind für dieses Beispiel ohne Bedeutung. Die Zeichen, die der AUSGABE-PROZESS ausgeben soll, müssen ihm von den anderen Prozessen in geeigneter Weise übergeben werden. Das Betriebssystem hat daraufhin den AUSGABE-PROZESS zu starten, damit er die Zeichen annehmen und bearbeiten kann.

Für die Übergabe von Informationen (hier: Zeichen für das Terminal) muß ein Multitasking-Betriebssystem ein Betriebsmittel bereitstellen, mit dessen Hilfe 'beliebig' viel Information (Daten) von jedem Prozeß an jeden Prozeß übergeben werden kann. Das Betriebssystem verwaltet dazu eine gewisse Anzahl von freien Speicherblöcken, die es bei Bedarf jedem Prozeß zur Verfügung stellen kann. Man nennt die Verwaltung von freien Speicherplätzen eine Poolverwaltung, wobei man unter Pool eine Liste freier Speicherblöcke versteht. Der freie Speicherblock wird dementsprechend auch 'Poolelement' genannt.

Da der Speicher jedes Rechners physikalisch in seiner Größe begrenzt ist, steht natürlich nicht 'beliebig' viel Speicher zur Verfügung. Die Hardware muß aber so dimensioniert sein, daß der RAM-Bereich für die zu erledigenden Aufgaben groß genug ist (vgl. Kapitel 5.1 Speicherabschätzung).

2.3.1 POOL-Elemente

Bei dieser Art der dynamischen Speicherzuweisung steht natürlich bei der Programmierung der Prozesse und auch des Betriebssystems noch nicht fest, welcher Prozeß welches Poolelement erhält. Daraus ergibt sich die Forderung, daß alle Poolelemente eines Pools den gleichen Regeln folgen müssen. Sie müssen also in Bezug auf Größe (Anzahl der Bytes), Strukturierung und Adressierung gleich sein. Der Pool ist in der Datenerklärung deshalb als 'Feld aus Strukturen' erklärt (Abb. 2-7).

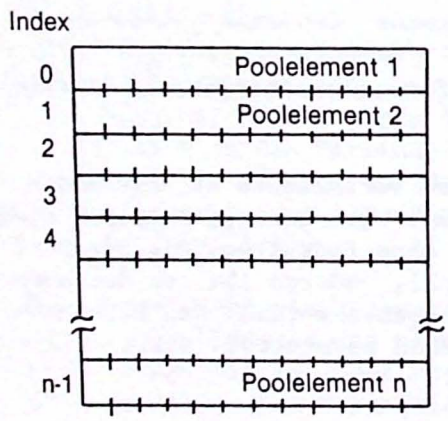

Abb. 2-7: Feld aus Poolelementen

Oftmals kommt es vor, daß von den Prozessen Poolelemente von stark unterschiedlicher Länge benötigt werden. Wegen der Speicherbegrenzung im System ist es meist nicht möglich, die Länge der Poolelememente gleich der maximal benötigten Länge zu machen. Das Produkt 'Anzahl der Poolelemente * Anzahl der Bytes pro Poolelement' ergibt den vom Pool belegten Speicherplatz. Durch

das Heraufsetzen der Anzahl der Bytes pro Poolelement müßte man die Anzahl der Poolelemente entsprechend senken, um mit dem gleichen Speicherplatz auszukommen. Da aber eine ausreichende Anzahl Poolelemente zur Verfügung stehen muß, funktioniert diese Methode oftmals nicht. Es werden deshalb in solchen Fällen vom Betriebssystem mehrere Pools mit unterschiedlicher Anzahl und unterschiedlicher Länge der Poolelemente geführt. Bei der Anforderung eines Poolelements muß der Anwender angeben, von welchem Pool er ein Element möchte. In dem Beispiel von Kapitel 2.3 wird ein Prozeß, wenn er kurze Ausgaben veranlassen möchte, die Zeichen in einem relativ kurzen Poolelement an den AUSGABE-PROZESS übergeben, während die Daten für eine längere Ausgabe in entsprechend größere Poolelemente verpackt werden. Man erreicht dadurch eine effektivere Speicherausnutzung.

In vielen Fällen kann man die Größe der Poolelemente nicht entsprechend der größten Anzahl, der von einem Prozeß zum anderen zu transferierenden Bytes, machen. Dies können unter Umständen mehrere KBytes sein. Hätte jedes Poolelement diese Größe, so wäre die Anzahl der zur Verfügung stehenden Poolelemente stark beschränkt. Der Anwender muß deshalb schon bei seiner Programmkonzeption beachten, daß er lange zusammenhängende Daten auf mehrere Poolelemente aufteilen muß. Für den Zusammenhang der Daten in den verschiedenen Poolelementen ist allein der Anwender verantwortlich.

Fordert ein Prozeß ein Poolelement an, so wird ihm vom Betriebssystem entweder der Index oder die absolute Adresse der Feldzeile übergeben. In einer höheren Programmiersprache erfolgt die Adressierung mit Index symbolisch, so daß sich für den Programmierer in dieser Beziehung keinerlei Probleme ergeben. Wird in Assembler programmiert, muß man auf die Anfangsadresse des Pools das Produnkt Index * Poolelementgröße addieren, um auf die Anfangsadresse des eigenen Poolelements zu kommen. Erhält man vom Betriebssystem die Adresse, was sicherlich von der Laufzeit günstiger ist, wird diese Aktion für den Anwender natürlich überflüssig. Das Betriebssystem hat sie bei der Initialisierung des Pools bereits durchgeführt. Abb. 2-8 verdeutlicht die Adressierung der Poolelemente.

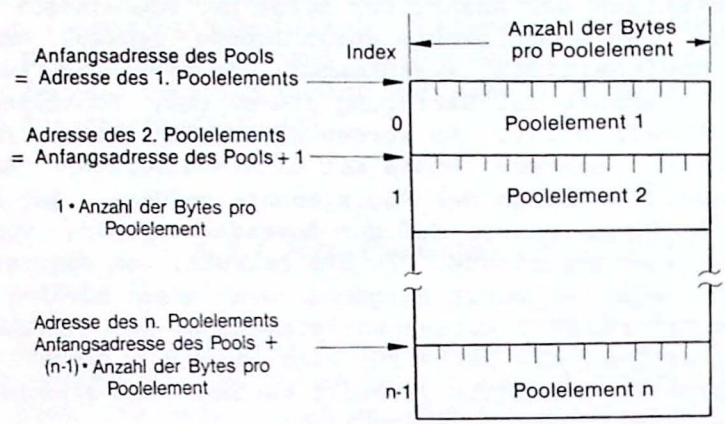

Abb. 2-8: Adressierung der Poolelemente

Ein Poolelement wird hauptsächlich dann verwendet, wenn Information von einem Prozeß an einen anderen Prozeß übergeben werden soll. Der Zielprozeß benötigt gewisse Angaben aus denen er ersehen kann, was die im Poolelement enthaltene Information für ihn bedeutet. Deshalb wird standardmäßig jedes Poolelement mit einer Anzahl Identifizierungsbytes versehen. In den meisten Anwendungsfällen werden von einem Prozeß mehrere Meldungsarten abgesetzt. Man versieht deshalb oft die Poolelemente mit den beiden Kennungen SOURCE_PID (Absender, PID = Prozeß-Identifikation) und EVENT_TYPE (Ereignis-Typ),(siehe Kapitel 7.3). Außerdem benötigen manche Betriebssysteme eine Information, an welchen Prozeß das Poolelement vergeben ist. Es wird deshalb noch das Kennbyte OWNER_PID (Eigentümeridentifikation) mitgegeben.

Abb. 2-9 zeigt die Information, die sinnvollerweise jedes Poolelement enthält. Dieser Poolelementkopf wird ergänzt von einer Anzahl freier Bytes, die der Anwender beliebig beschreiben darf. Die Anzahl der freien Bytes ist abhängig vom Anwendungsfall und wird in der Systemkonzeption festgelegt.

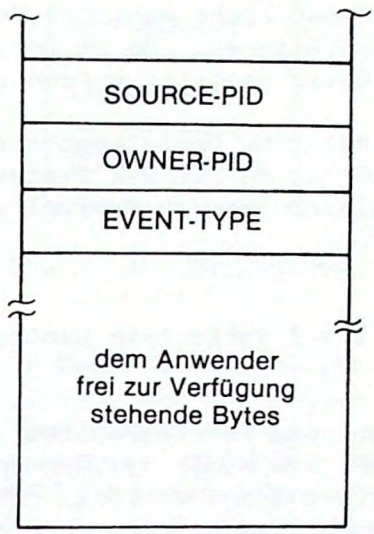

Abb. 2-9: Strukturierung eines Poolelements

Bisher haben wir immer davon gesprochen, daß der Anwender Poolelemente vom Betriebssystem anfordert. Das Betriebssystem muß also ein Mittel bereitstellen, das dem Anwender ein Poolelement zur freien Verfügung übergibt und mit dessen Hilfe ein Poolelement, das nicht mehr benötigt wird, wieder in den vom Betriebssystem verwalteten Pool zurückgegeben werden kann. Die Rückgabe ist unbedingt erforderlich, da natürlich nur eine begrenzte Anzahl zur Verfügung steht und sonst der Pool bald leer wäre.

Das Mittel besteht in den sogenannten Betriebssystem-Prozeduren, die der Anwender aufruft, wenn er die Dienste des Betriebssystems in Anspruch nehmen möchte. Für die Anforderung von einem Poolelement steht die Prozedur 'GET_POOL' zur Verfügung. Sie übergibt dem Aufrufer den Index oder die Adresse des ihm nun zur Verfügung stehenden Poolelements. Als Parameter gibt der Anwender an, von welchem Pool er ein Element möchte.

Will ein Anwender ein Poolelement an den Pool zurückgeben, so ruft er die Prozedur 'RELEASE_POOL' auf. Als Parameter muß er den Index oder die Adresse des zurückzugebenden Poolelements übergeben. Außerdem gibt er an, an welchen Pool die Rückgabe

erfolgen soll. Natürlich darf er das Poolelement nur an den Pool zurückgeben, von dem er es erhalten hat. Mit dem Prozeduraufruf steht ihm das Poolelement nicht mehr zur Verfügung. Er darf es weder auslesen noch beschreiben, da es bereits wieder einem anderen Prozeß zur Verfügung gestellt worden sein kann.

Kommt es einmal vor, daß mehr Poolelemente angefordert wurden als überhaupt vorhanden waren, so ist das System nicht mehr vollständig lebensfähig. Vergleiche hierzu Kapitel 2.9.

2.3.2 Verkettete Listen

Die Poolelemente werden vom Betriebssystem in einer Warteschlange verwaltet. In Kapitel 2.2 wurde das Grundprinzip der Queues anhand einer Prozeßwarteschlange erklärt. Hier soll nun gezeigt werden, daß das Warteschlangenprinzip auch bei der Verkettung der Poolelemente verwendet wird. Dazu muß nun näher auf die softwaremäßige Realisierung einer Warteschlange eingegangen werden.

Eine Queue besteht aus einem Queue-Header (QHDR, Warteschlangenkopf) und einer Anzahl von geketteten Speicherelementen. Der Queueheader beinhaltet je einen Zeiger auf das erste und letzte Element der Queue, sowie einem Zähler, in dem die Anzahl der in der Queue befindlichen Speicherstücke steht (Abb. 2-10). Je nachdem, ob in dem jeweiligen Betriebssystem mit Indizes oder absoluten Adressen gearbeitet wird, steht in den Zeigern der Index bzw. die absolute Adresse der entsprechenden Poolelemente.

Die Methode, gleich die absoluten Adressen abzuspeichern, ist die am meisten verbreitete und auch laufzeitmäßig am günstigsten, weil sonst vor jedem Zugriff auf ein Poolelement die Adresse aus der Anfangsadresse des Pools, dem Index des Poolelements und der Länge eines Poolelements ausgerechnet werden müßte. Im weiteren Verlauf der Beschreibung wird deshalb nur noch auf diese Methode eingegangen.

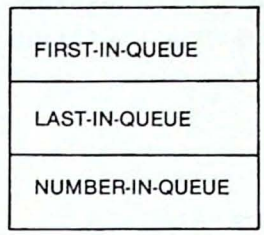

Abb. 2-10: QUEUEHEADER

Der Zeiger auf das erste Queue-Element heißt FIRST_IN_QUEUE, der auf das letzte LAST_IN_QUEUE, und der Zähler wird NUMBER_IN_QUEUE genannt. Enthält die Queue kein Element, so ist dies im Queue-Header dadurch gekennzeichnet, daß FIRST_IN_QUEUE und LAST_IN-_QUEUE mit einer sog. NULL_ID beschrieben sind. Darunter ist eine Bitkombination zu verstehen, die keinem gültigen Wert entspricht (z.B. FFFFH:FFFFH als 8086-Adresse). In NUMBER_IN_QUEUE steht dann 0. Wird nun in diese Queue ein Poolelement eingekettet, so steht in FIRST_IN_QUEUE die Adresse dieses Speicherelements. Da dieses Element gleichzeitig auch das letzte innerhalb der Queue ist, muß in LAST_IN_QUEUE ebenfalls diese Adresse stehen (Abb. 2-11). NUMBER_IN_QUEUE steht auf 1.

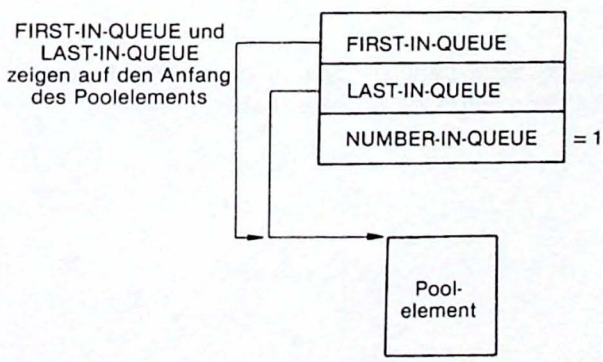

Abb. 2-11: Ein Element in der Queue

Kommt nun ein weiteres Poolelement hinzu, so bleibt FIRST_IN-_QUEUE unverändert auf dem ersten stehen. In LAST_IN_QUEUE steht jetzt die Adresse des zweiten (letzten) Elements (Abb. 2-12). NUMBER_IN_QUEUE steht auf 2.

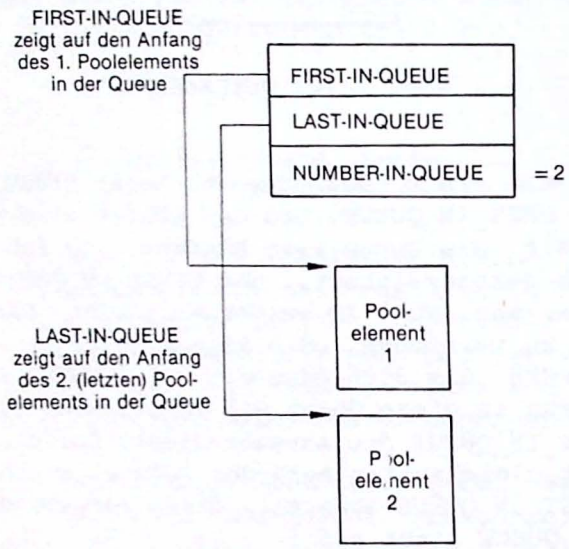

Abb. 2-12: Zwei Elemente in der Queue

Bei 3 und mehr Elementen in der Queue kommt man mit dem Queueheader allein nicht mehr aus (Abb. 2-13).

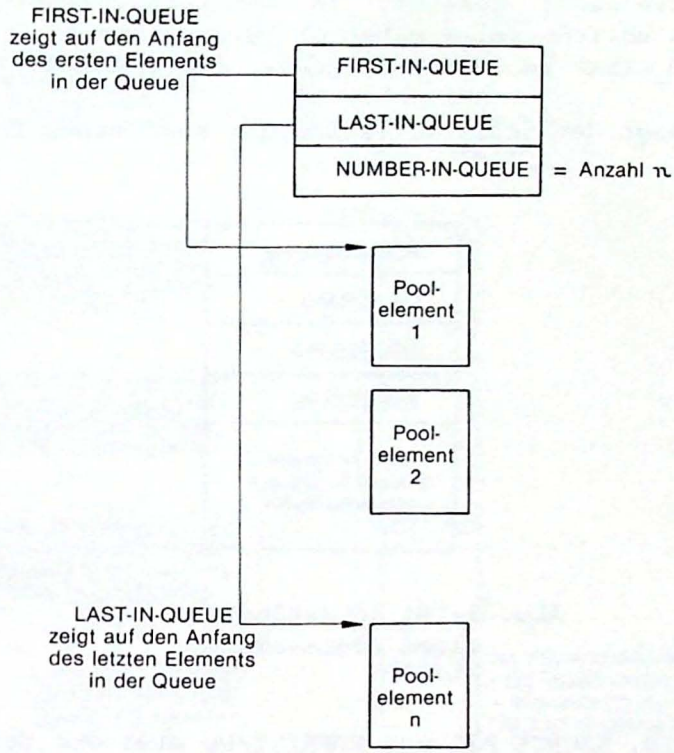

Abb. 2-13: n-Elemente in der Queue

FIRST_IN_QUEUE zeigt auf das erste, LAST_IN_QUEUE auf das letzte und in NUMBER_IN_QUEUE steht die Anzahl der Elemente. Welches Poolelement als zweites in der Queue steht, würde man so nicht feststellen können. Mit dem dritten, vierten, etc. wäre es ebenso, nur das letzte steht im QHDR und ist damit bekannt.

Es muß also in dem ersten Queueelement ein Zeiger vorhanden sein, der auf das in der Queue folgende Element hinweist. Ein Zeiger im zweiten Queueelement weist auf das dritte usw. Der Zeiger im vorletzten Queueelement zeigt auf das letzte. Dessen Zeiger enthält die NULL_ID als Kennzeichen, daß es kein nachfolgendes Queueelement gibt.

Der Zeiger auf das folgende Queueelement wird FORWARD_LINK (Vorwärtsverkettung) genannt. In Betriebssystemen werden die Poolelemente üblicherweise gekettet (Grundprinzip). Deshalb wird der FORWARD_LINK in den Kopf jedes Poolelements aufgenommen.

Abb. 2-14 zeigt den jetzt vollständigen Kopf eines Poolelements.

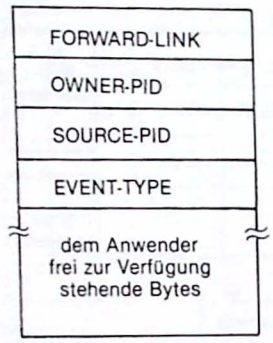

Abb. 2-14: Vollständiger Kopf eines Poolelements

Vor OWNER_PID, SOURCE_PID und EVENT_TYPE wird die Vorwärtsverkettung eingetragen. Das Beschreiben des FORWARD_LINK's und der beiden PIDs obliegt ausschließlich dem Betriebssystem.

In Abb. 2-15 ist eine vollständige Warteschlange mit vier Poolelementen dargestellt. Mit dieser Warteschlange können beide Prinzipien (FIFO bzw. LIFO) je nach Anwendungsfall realisiert werden.

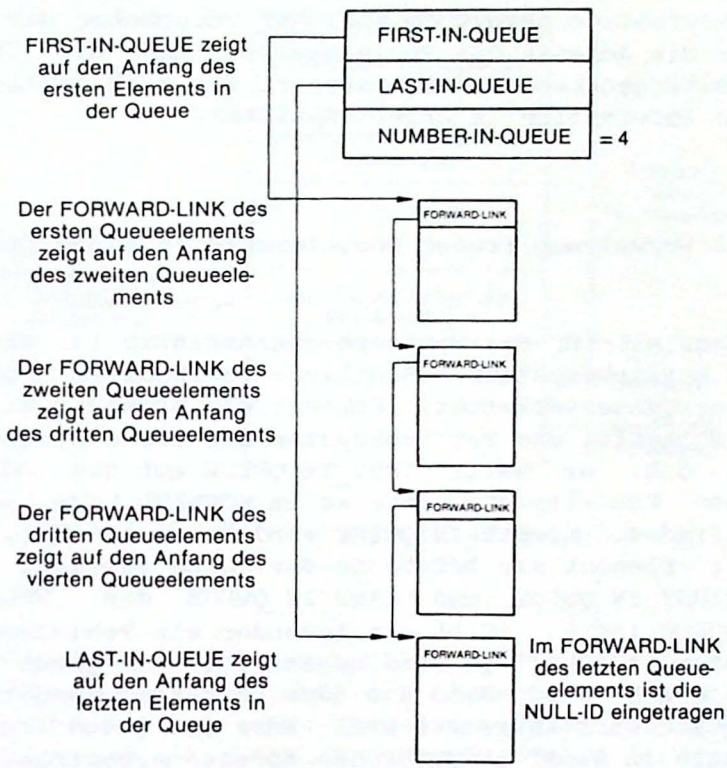

Abb. 2-15: Verkettung in einer Queue mit vier Elementen

Beim Ausketten nimmt das Betriebssystem immer das erste Element der Queue.

Werden neue Queueelemente immer ans Ende der Warteschlange eingekettet, so wird vom Betriebssystem immer das am längsten in der Queue befindliche Element ausgekettet. Das entspricht dem FIFO-Prinzip.

Werden dagegen neue Elemente immer an den Anfang eingekettet, so holt das Betriebssystem das zuletzt eingekettete Element als erstes ab, was die Realisierung des LIFO-Prinzips bedeutet.

Die Kettbarkeit der Poolelemente ist deshalb so wichtig, weil bei der Weitergabe von Information in Multitasking-Betriebssystemen nicht die Information selbst im Speicher verschoben wird, sondern es wird nur die Adresse des Poolelements, das die Information enthält, weitergegeben. Dadurch erspart man sich zeitaufwendiges Kopieren von Information im Arbeitsspeicher.

2.3.3 Verwaltung freier Poolelemente in einer Queue

Ein Anwendungsfall für das Warteschlangenprinzip ist die Poolverwaltung des Betriebssystems. Sämtliche verfügbaren Poolelemente sind in einer Queue verkettet. Fordert ein Anwender ein Poolelement an, so kettet das Betriebssystem das erste Element aus der Queue aus, d.h. es setzt FIRST_IN_QUEUE auf die Adresse des nachfolgenden Poolelements, die es im FORWARD_LINK des ersten Elementes findet. NUMBER_IN_QUEUE wird dekrementiert. Wäre das ausgekettete Element das letzte in der Queue gewesen, so steht nun in FIRST_IN_QUEUE und LAST_IN_QUEUE die NULL_ID und NUMBER_IN_QUEUE ist 0. Gibt ein Anwender ein Poolelement an das Betriebssystem zurück, so wird es ans Ende der Queue angehängt. In LAST_IN_QUEUE wird dazu die neue Adresse eingetragen und NUMBER_IN_QUEUE wird inkrementiert. Wäre die Queue leer gewesen, so müßte auch in FIRST_IN_QUEUE die Adresse eingetragen werden. In diesem Fall wird die Poolverwaltung als FIFO aufgebaut. Man könnte es aber ebenso als LIFO realisieren. Vor- bzw. Nachteile des einen Verfahrens gegenüber dem anderen gibt es bei dieser Anwendung nicht, da hier der zeitliche Bezug des Eintrags verschiedener Poolelemente in die Queue nicht relevant ist.

2.3.4 Verwaltung freier Poolelemente in einer Liste

Eine andere Möglichkeit der Poolverwaltung besteht darin, daß die Adressen der freien Poolelemente in einer Liste (Feld aus Adressen) eingetragen sind (Abb. 2-16).

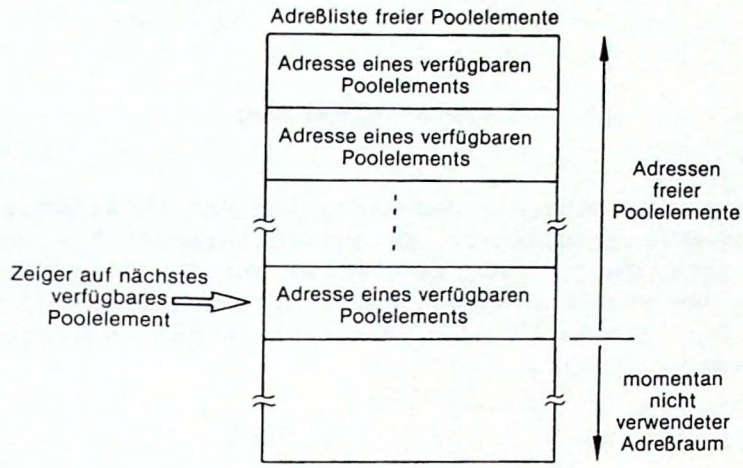

Abb. 2-16: Verwaltung freier Poolelemete in einer Liste

Das Betriebssystem führt einen Zeiger, der auf die nächste verfügbare Adresse zeigt. Fordert ein Anwender ein Poolelement an, so übergibt ihm das Betriebssystem die Adresse aus der Liste, auf die sein Zeiger gerade zeigt. Anschließend wird der Zeiger dekrementiert. Gibt ein Anwender ein Poolelement zurück, so inkrementiert das Betriebssystem zuerst den Zeiger und schreibt dann auf den nun vom Zeiger adressierten Speicherplatz die Adresse des zurückgegebenen Poolelements.

Der Vorteil dieser Methode liegt in der günstigeren Laufzeit der Betriebssystemaufrufe. Die Vergabe und Rückgabe der Poolelemente ist ca. 50% schneller, als wenn die Poolelememte bereits im Pool verkettet sind. Der Nachteil ist, daß die Liste mehr Datenspeicher benötigt als ein Queueheader.

Für die Adresse jedes Poolelements muß in der Liste der Platz bereitgestellt werden. Beim Prozessor 8086 besteht die Adresse aus je 2 Byte Segment und Offset. Das Segment ist für alle Poolelemente gleich und muß deshalb nicht in der Liste abgelegt werden, es werden nur die 2 Byte Offset eingetragen. Bei 250 Poolelementen wären das 500 Byte Datenspeicher gegenüber nur 5 Byte für einen QueueHeader. Es liegt deshalb nahe, diese Art der

Poolverwaltung nur dann einzusetzen, wenn ausreichend Datenspeicherkapazität zur verfügung steht.

2.4 Stack-Verwaltung

Der Stack hat eine zentrale Bedeutung für das Funktionieren eines Multitasking-Betriebssystems. Es erfolgt deshalb hier eine ausführliche Behandlung, bei der zuerst auf die allgemeine Funktionsweise des Stacks eingegangen wird. Anschließend wird die Bedeutung des Stacks für die Prozesse eines Multitasking-Betriebssystems erläutert.

2.4.1 Funktionsweise des Stacks

Die Funktionsweise des Stackmechanismuses beim Intel-Prozessor 8086/8088 soll anhand eines Prozeduraufrufs erläutert werden. Zur Adressierung des Stacks verwenden die Intel-Prozessoren die Register Stacksegment (SS), Stackpointer (SP) und Basepointer (BP). Die absolute Adresse errechnet sich aus SS * 10H(exadezimal) + SP bzw. SS * 10H + BP. SS enthält somit die Segmentadresse des Stacks.

Ein Segment kann bis zu 64 kBytes lang sein, das ist auch die maximal mögliche Größe eines Stackbereichs. Der Stackpointer enthält den Offset des Stacks, er zeigt immer auf die letzte beschriebene Stackzelle. Mit dem Basepointer werden Parameter und lokale Variable in Prozeduren adressiert. Der Stack des 8086/8088 ist 16 Bit breit, es wird immer ein '16-Bit-Wort' auf den Stack geschrieben bzw. vom Stack gelesen. Vor dem Beschreiben wird der Stackpointer um 2 dekrementiert, nach dem Lesen wird er um 2 inkrementiert.

Nun zu dem Beispiel:

Abb. 2-17 zeigt den Stack vor Beginn eines Prozeduraufrufs. SP zeigt auf die letzte beschriebene Stackzelle, die Stellung von BP ist hier ohne Bedeutung.

Beim Prozeduraufruf werden als erstes die Parameter auf dem Stack abgelegt. Anschließend wird der 'CALL'-Befehl abgesetzt. Mit ihm wird die Rücksprungadresse (Register CS und IP) auf dem Stack gespeichert. Abb. 2-18 zeigt den Stack nach dem 'CALL'.

Im Prozedur-Prolog wird, sofern die Prozedur 'PUBLIC' ist, das Datensegment (DS) auf dem Stack gespeichert. Anschließend wird der Basepointer darauf abgelegt und SP in BP geladen. Mit BP können somit die Parameter und die lokalen Variablen adressiert werden. BP wird während des Prozedurablaufs nicht mehr verändert. Danach wird der Stackpointer um die Anzahl der lokalen Variablen dekrementiert (Abb. 2-19). Nun kann mit der Abarbeitung der Prozedur begonnen werden.

Vor der Rückkehr aus der Prozedur wird BP in SP geladen, anschließend wird der ursprüngliche Inhalt von BP und DS wieder vom Stack übernommen (Abb. 2-20).

Bei der Rückkehr aus der Prozedur ('RET'-Befehl) wird die alte Adresse (CS und IP) wieder vom Stack übernommen. SP wird um die Anzahl der Parameter inkrementiert. Der Stack hat somit wieder das Aussehen von Abb. 2-17.

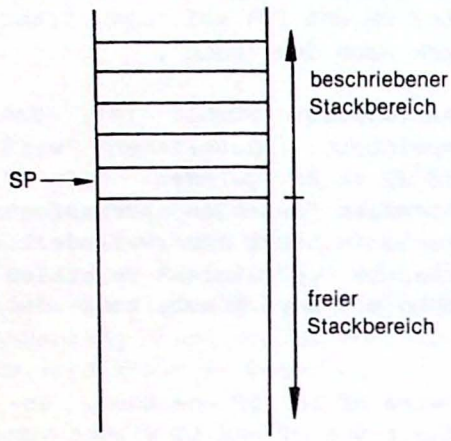

Abb. 2-17: Stack vor dem Prozeduraufruf

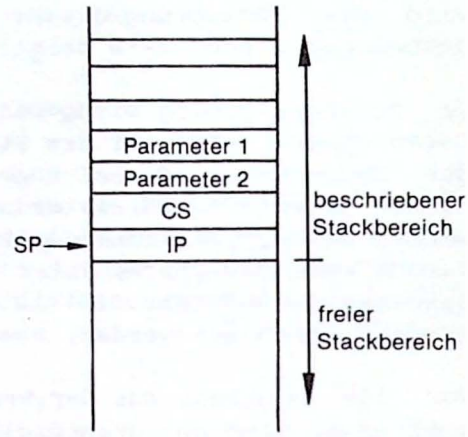

Abb.2-18: Stack nach dem Prozeduraufruf

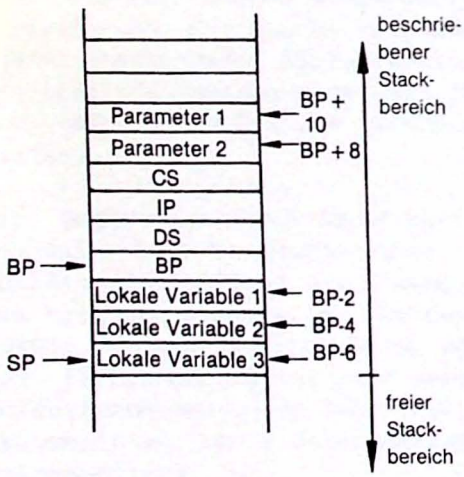

Abb. 2-19: Stack nach dem Prozedurprolog

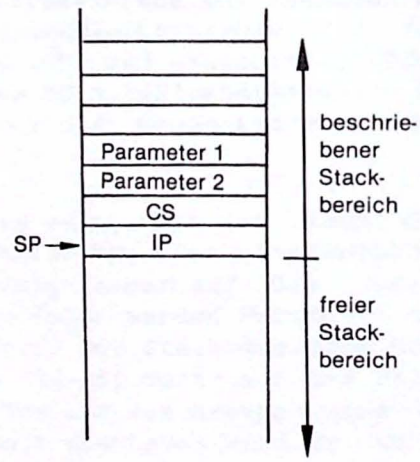

Abb. 2-19: Stack nach dem Prozedurepilog

2.4.2 Der Stack im Betriebssystem

In einem Multitasking-Betriebssystem hat jeder Prozeß seinen eigenen Stack. Ein Prozeßwechsel kann nur über das Betriebssystem erfolgen. Den Anreiz hierfür liefert entweder ein Hardware-Interrupt (HW-Timer, HW-Ereignis) oder der gerade laufende Prozeß benutzt eine Betriebssystem-Routine. Betriebssystem-Routinen sind Prozeduren, die vom Anwender z.B. durch einen 'CALL'-Befehl aufgerufen werden. Eine andere Methode ist die, sog. 'Software-Interrupts' zu benutzen (s. 5.1.4.2), was aber für den Stackmechanismus ohne Belang ist. Bei den INTEL-Prozessoren 8086/8088 wird beim'CALL' der Befehlszähler auf dem Stack abgespeichert. Bei Interrupts wird neben CS und IP auch das Flagregister auf dem Stack abgelegt. In den Interruptprogrammen müssen als erstes die Register gespeichert werden. Dies geschieht mittels 'PUSH'-Befehlen, die ebenfalls auf den Stack arbeiten. Will das Betriebssystem also einen Prozeßwechsel vornehmen, so muß es die Stackadressen (SS, SP) des neuen Prozesses laden. Vorher muß es die Stackregister des zu verdrängenden Prozesses abspeichern, damit es bei der Rückkehr zu diesem Prozeß wieder richtig aufsetzen kann. Der Grund dafür liegt darin, daß die Prozesse unabhängig voneinander sind, d.h. beim Prozeßwechsel darf nicht mehr auf den Stack des alten Prozesses gearbeitet werden. Die Rücksprungadresse aus der Betriebssystem-Prozedur liegt auf dem Stack des neuen Prozesses.

Ein Beispiel soll dieses Vorgehen verdeutlichen:

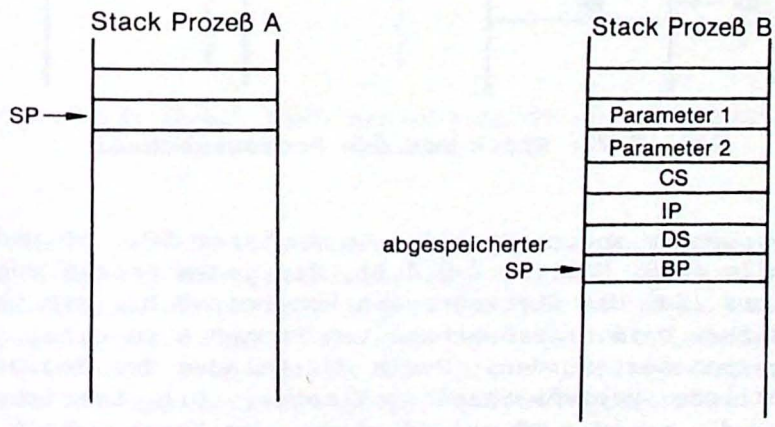

Abb. 2-21: Stack der Prozesse A und B vor dem Interrupt

Abb. 2-21 zeigt den Stack der beiden Prozesse A und B, Prozeß A ist gerade in Bearbeitung, von Prozeß B wurden die Stackadressen (SS und SP) abgespeichert. SP von Prozeß A zeigt auf die letzte benutzte Stackzelle. Davor liegen irgendwelche lokalen Variablen und Adressen, die für dieses Beispiel ohne Bedeutung sind. Der Prozeß wird nun durch einen Hardware-Interrupt unterbrochen. Der Prozessor legt daraufhin Flags und Befehlszähler (CS und IP) auf den Stack. Anschließend werden die Register gespeichert. In dem Interruptprogramm wird nun eine Betriebssystem-Routine mit zwei Parametern aufgerufen, die einen Prozeßwechsel veranlaßt. Durch diesen Prozeduraufruf werden wiederum der aktuelle Befehlszähler und anschließend die Register DS und BP auf dem Stack abgelegt (Abb. 2-22).

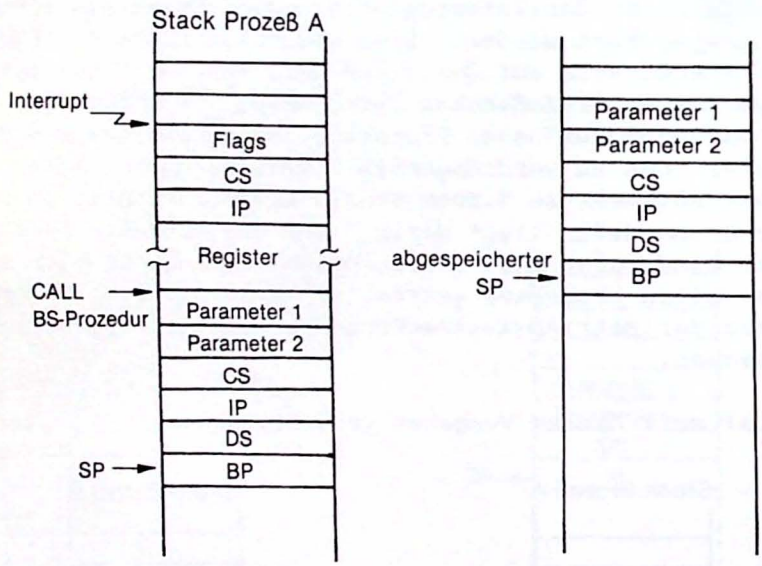

Abb. 2-22: Stack vor dem Prozesswechsel

Das Betriebssystem speichert sich nun Register SS, SP und BP in einer Tabelle (PCB, Kapitel 2.6.4.1), die jedem Prozeß zugeordnet ist, ab und lädt die Stackadressen von Prozeß B, die auf die gleiche Weise beim Prozeßwechsel von Prozeß B zu einem anderen Prozeß abgespeichert wurden. Durch das Umladen der Register ist der eigentliche Prozeßwechsel vollzogen. Die Betriebssystem-Routine wird, nachdem BP und DS wieder vom Stack geholt wurden, durch den 'RET'-Befehl beendet. Der Prozessor holt sich beginnend

bei SP die nächsten zwei 16-Bit-Worte vom Stack, jetzt aber von Prozeß B, da SS und SP umgeladen wurde, und lädt sie in seinen Befehlszähler (CS und IP). Der Befehlszähler zeigt damit auf den Code von Prozeß B und die Stackregister SS und SP arbeiten auf den zu Prozeß B gehörenden Stackbereich. Die Betriebssystem-Prozedur ist damit beendet, es wird der Code von Prozeß B bearbeitet.

Ein separates Abspeichern und Laden des Datensegments entfällt, da es im Prozedurprolog auf dem Stack gespeichert wird. Im Prozedurepilog wird es wieder vom Stack übernommen. Abb. 2-23 zeigt die Stacks der Prozesse nach dem 'RET'-Befehl.

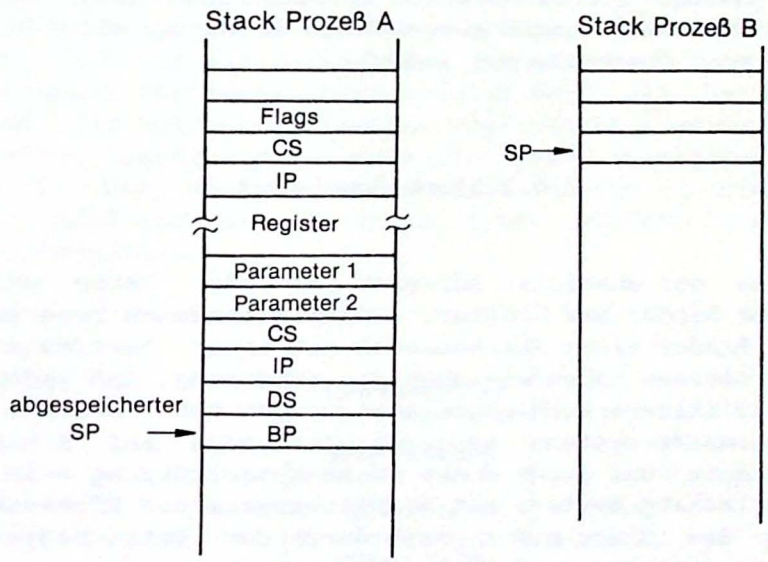

Abb. 2-23: Stack nach Beendigung der Betriebssystem-Prozedur

Der Wechsel zurück zu Prozeß A erfolgt in der gleichen Weise. Nach dem Aufruf der entsprechenden Betriebssystem-Routine wird die Stackadresse von Prozeß A geladen. Der SP steht wieder an der Stelle wie in Abb. 2-22. Mit dem 'RET' der Betriebssystem-Prozedur wird CS und IP wieder in den Befehlszähler übernommen. SP zeigt nun auf das letzte abgespeicherte Register des Interruptprogrammes, das nun mit 'POP'-Befehlen beendet wird. Der anschließende Befehl 'IRET' übernimmt wieder CS,IP und die Flags

von Prozeß A. Befehlszähler, Stack und Register haben somit wieder den Stand vor dem Interrupt (Abb. 2-21). Daß dazwischen Prozeß B (und u.U. weitere Prozesse) abgelaufen ist, störte den Ablauf von Prozeß A nicht.

Erfolgt ein Prozeßwechsel, indem ein Prozeß eine Betriebssystem-Prozedur verwendet, so ist die Funktionsweise die gleiche wie bei der Verdrängung durch einen Interrupt. Es entfällt nur das Speichern aller Register einschließlich des Flagregisters.

Kurz zusammengefaßt bedeutet das:

Um einen Prozeßwechsel zu vollziehen, muß der Scheduler den Inhalt der PROCESS_QUEUEs bewerten (Kapitel 2.6.4.2). Damit wird der neu zu startende Prozeß ausgewählt. Um ihn zu starten, muß er nur noch dessen Stackadressen laden.

2.4.3 Stack-Zuweisung

Die Vergabe der absoluten Adressen für Code, Daten und Stack erfolgt vom Binder bzw Locater. Jedem gebundenen Programmsystem weist der Binder einen Stackbereich mit einer bestimmten Größe zu. Bei Prozessen haben wir aber die Forderung, daß jedem Prozeß ein eigener Stackbereich zugewiesen werden muß. Übliche Binder für Mikrocomputersysteme basieren meist noch auf Ein-Benutzer Konfigurationen und geben daher keine Unterstützung beim Binden eines Multitasking-Systems mit Betriebssystem und Prozessen. Die Verteilung des Stack muß deshalb durch das Betriebssystem bei dessen Initialisierung erfolgen.

Im Betriebssystem-Modul ist ein zusammenhängender Datenspeicherbereich als sog. 'STACK-ARRAY' erklärt. Die Größe errechnet sich aus der Anzahl der Prozesse multipliziert mit der Stackgröße eines Prozesses. In der Regel obliegt es dem Systemprogrammierer das STACK_ARRAY an die einzelnen Prozesse zu verteilen. Das Betriebssystem bereitet beim Hoch-lauf die Stackbereiche der Prozesse so auf, daß vom Scheduler die Anfangsadressen geladen werden können, und die Prozesse so nacheinander gestartet werden. Ausführlich wird auf die Betriebssystem-Initialisierung und den Prozeßstart in Kapitel 3.2.2 eingegangen.

2.5 Prozeß-Kommunikation

Wie in den vorangegangenen Kapiteln bereits angedeutet, besteht in Realtime-Betriebssystemen die Notwendigkeit, daß die Prozesse untereinander Informationen (Nachrichten) austauschen. Der Nachrichtenaustausch ist ein grundlegendes Hilfsmittel für Systeme mit Prozessen und dient zur Entkopplung der Prozesse. D.h. dadurch wird es sehr stark erleichtert, die verschiedenen Funktionen eines Systems, die gegenseitige Abhängigkeiten aufweisen, auf mehrere Prozesse aufzuteilen.

Um Mißverständnissen vorzubeugen sei hier darauf hingewiesen,daß dies in erster Linie für Multitask-Realzeit-Systeme zutrifft. Andere Betriebssysteme, die z.B. nach der 'Timesharing'-Methode arbeiten, sind von der Betrachtung ausgenommen.
Der Austausch der Nachrichten erfolgt über das Betriebssystem. Auch hat das Betriebssystem die Möglichkeit, Meldungen an die Prozesse zu senden (z.B. wenn ein Timer abgelaufen ist, Kapitel 2.7). Der Nachrichtenaustausch erfolgt in Event-Blöcken. Darunter sind speziell für diesen Zweck strukturierte Poolelemente zu verstehen.

Abb. 2-24: Event-Block

Abb. 2-24 zeigt ein derartig strukturiertes Poolelement. Da Poolelemente kettbar sind, ist auch hier ein FORWARD_LINK vorhanden. Der eigentlichen Meldung, die mit dem Betriebssystem nichts mehr zu tun hat und die wiederum je nach Anwendungsfall besonders

strukturiert sein kann (z.B. in Jobcode und Parameter), sind die Empfangs-Prozeß-Nr., die Absender-Prozeß-Nr. und der EVENT_TYPE vorangestellt. Empfänger und Absender dienen oftmals dem Betriebssystem zur Überwachung und werden auch von ihm beschrieben (Kapitel 2.9). Der Anwender hätte die Möglichkeit diese Felder zu bewerten, um z.B. je nach Absender, auf die Nachricht zu reagieren. Im Interesse einer Trennung der Zuständigkeiten von Betriebssystem und Anwender und der daraus resultierenden Fehlerwahrscheinlichkeit (siehe Kapitel 7.3.2.1) wird hiervon jedoch dringend abgeraten. Der EVENT_TYPE wird vom Anwender beschrieben. Er hat hierdurch die Möglichkeit seine Meldungen in Gruppen zu unterteilen, er muß es aber nicht.

Versendet das Betriebssystem die Meldung, daß ein Timer oder ein Echtzeitauftrag abgelaufen ist, so trägt es im EVENT_TYPE eine Kennung ein, daß es sich bei der Meldung um dieses Ereignis handelt. In den Meldungsteil wird dann TIMER_TYPE und USER_CODE eingetragen, was den Empfänger in die Lage versetzt die Nachricht exakt aufzuschlüsseln. (Die Beschreibung des Timers und der damit zusammenhängenden Namen erfolgt in Kapitel 2.7)

2.5.1. Briefkasten-Prinzip ('Mailbox')

Jeder Prozeß hat zum Zwischenspeichern der an ihn adressierten Nachrichten einen 'Briefkasten'. In ihm laufen alle Nachrichten auf, bis das Betriebssystem den Prozeß startet und er seinen Briefkasten zu leeren beginnt. Eine einfache Möglichkeit, die Zusammengehörigkeit und die zeitliche Reihenfolge von Meldungen zu erhalten, ist, sie in eine Warteschlange einzuketten. Deshalb ist jedem Prozeß eine Nachrichtenwarteschlange (EVENT_QUEUE) zugeordnet.

Hat das Betriebssystem eine Nachricht an einen Prozeß zu versenden, so kettet es in der Regel das Poolelement mit der Meldung an das Ende der EVENT_QUEUE des Empfangsprozesses ein (Abb. 2-25) (FIFO-Prinzip, Kapitel 2.2.1). Die Meldungen werden also, gemäß ihrer zeitlichen Reihenfolge in die Warteschlange eingekettet und später auch in dieser Reihenfolge abgearbeitet.

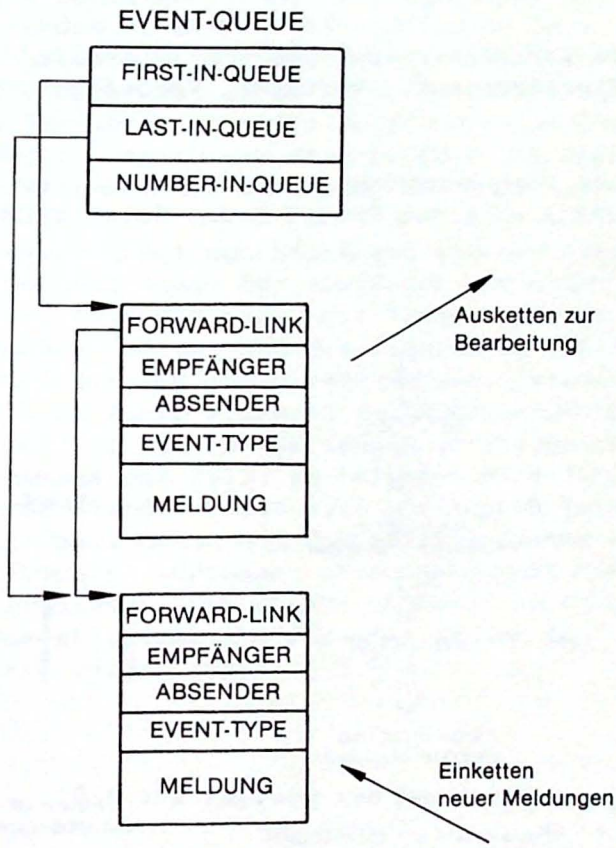

Abb. 2-25: Event-Queue

Prinzipiell besteht auch die Möglichkeit, die Meldung an den Anfang der EVENT_QUEUE zu setzen. Da dem Prozeß, wenn er bereit ist eine Meldung zu empfangen, immer die jeweils erste Meldung der EVENT_QUEUE ausgekettet wird, könnte man somit Nachrichten priorisieren (LIFO-Prinzip, Kapitel 2.2.2).

2.5.1.1 Versenden von Nachrichten

In den folgenden Kapiteln werden nun die in Kapitel 2.1.1 erklärten Begriffe (Prozeßzustand, Vertagen, Verdrängen, etc.) verwendet.

Die vollständige Vorgehensweise beim Versenden einer Nachricht von einem Prozeß A an einen Prozeß B ist folgendermaßen (Abb. 2-26):

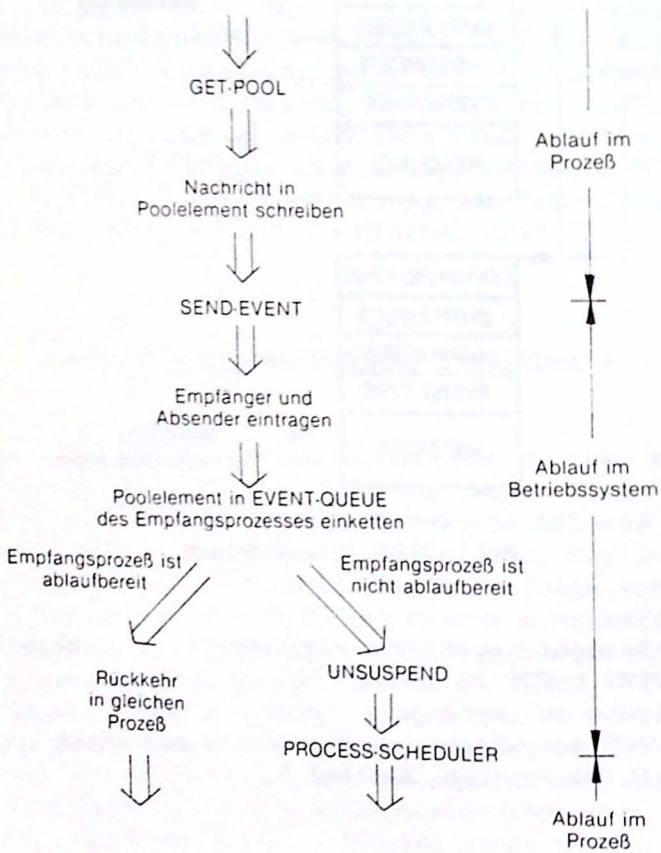

Abb. 2-26: Ablauf beim Versenden einer Nachricht

Prozeß A fordert vom Betriebssystem durch den Prozeduraufruf 'GET_POOL' ein Poolelement an. Er legt seine Nachricht im Poolelement ab, indem er EVENT_TYPE und das je nach Anwendungsfall strukturierte Datenfeld beschreibt. Anschließend veranlaßt er das Versenden der Nachricht durch den Betriebssytemaufruf 'SEND_EVENT'. Als Parameter übergibt er die Adresse des zu versendenden Poolelements, sowie die Prozeßnummer des Empfangsprozesses. Damit ist für Prozeß A das Versenden der Nachricht abgeschlossen.

Das Betriebssystem trägt den Empfänger und den Absender in die zu versendende Meldung ein. Der Empfänger wurde ihm als Parameter übergeben, der Absender ist die Nummer des gerade laufenden Prozesses. Danach kettet das Betriebssystem den Event-Block in die EVENT_QUEUE des Empfangsprozesses ein. Wurde der Empfangsprozeß bereits zu einem früheren Zeitpunkt ablaufbereit gemacht (Kapitel 2.6.3), so ist das Versenden der Nachricht abgeschlossen. Ist dies nicht der Fall, so befindet sich der Empfangsprozeß in einem Wartezustand. Falls sich der Prozeß im Zustand 'Warten auf Nachricht' befindet, muß das Betriebssystem ihn in den Zustand 'ablaufbereit' bringen. Dies geschieht mittels der Betriebssystem-Prozedur 'UNSUSPEND' und wird im Kapitel 2.6.4 beschrieben. Nur ablaufbereite Prozesse können vom 'PROCESS_SCHEDULER' gestartet werden (Kapitel 2.6.4.2).

2.5.1.2 Empfang von Nachrichten

Ein Prozeß im Zustand 'Running', der bereit ist eine Nachricht zu empfangen, ruft die Betriebssystem-Routine 'RECEIVE_EVENT' auf. Als Parameter übergibt er die Adresse einer Speicherzelle, auf die das Betriebssystem die Adresse des empfangenen Event-Blocks schreiben soll. Das Betriebssystem schaut daraufhin in die EVENT_QUEUE des betreffenden Prozesses. Sind Poolelemente eingekettet, so wird das erste ausgekettet. Die Adresse wird auf den vom Anwender übergebenen Zeiger geschrieben. Damit ist für das Betriebssystem der Empfang einer Nachricht abgeschlossen.

Es bleibt allein dem Anwender überlassen, wie die Nachricht ausgewertet wird und was mit dem Poolelement geschieht (z.B. Rückgabe an Pool, Weiterverwendung für Nachricht an anderen Prozeß, etc.).

Ist in der EVENT_QUEUE keine Nachricht enthalten, so vertagt das Betriebssystem diesen Prozeß. D.h., durch die Betriebssystem-Prozedur 'SUSPEND' wird der Prozeß in den Zustand 'Warten auf Nachricht' gebracht (Kapitel 2.6.3). Anschließend wird der PROCESS_SCHEDULER aufgerufen, der einen anderen Prozeß, der sich im Zustand 'Ablaufbereit' befindet, startet (Kapitel 2.6.4.2). Wie in Kapitel 2.5.1.1 beschrieben, versetzt das Absenden einer Nachricht an diesen Prozeß, den Prozeß wieder in den Zustand 'Ablaufbereit' und er kann danach vom PROCESS_SCHEDULER erneut gestartet werden. Er wird an der Stelle fortgesetzt, an der er vorher vertagt wurde, nämlich in der Prozedur 'RECEIVE_EVENT'. D.h. diese Prozedur kann jetzt ihren Auftrag erfüllen und kettet das in der EVENT_QUEUE jetzt vorhandene Poolelement aus. Wie oben beschrieben, wird es dem Anwender übergeben.

2.5.2 Warten auf Unterbrechung

Den Anlaß, daß ein Prozeß gestartet werden soll, bildet z.B. die Nachricht von einem anderen Prozeß. Oftmals werden aber bestimmte Ereignisse durch einen Hardware-Interrupt signalisiert. In dem Beispiel von Kapitel 2.3 wird dem AUSGABE-PROZESS vom USART durch einen Interrupt signalisiert, daß er für das nächste Zeichen aufnahmebereit ist. Das Betriebssystem hat daraufhin den Prozeß zu starten, der weitere Aktionen veranlaßt.

Erwartet ein Prozeß einen Interrupt und möchte er sich bis zu dessen Eintreffen vertagen, so ruft er die Betriebssystem-Prozedur 'WAIT_INTERRUPT' auf. Das Betriebssystem setzt ihn daraufhin durch seine Prozedur 'SUSPEND' in den Zustand 'Warten auf Interrupt' und startet über den PROCESS_SCHEDULER den nächsten auf Ablauf wartenden Prozeß. Trifft der Interrupt ein, so wird, (beim Prozessortyp 8086/8088) adressiert über die Interrupt-Vektor-Tabelle (s. 5.1.4), eine Interrupt-Prozedur gestartet ('Interrupt-Handler', s. 6.2.4.1). Diese Routine, sie gehört nicht zum Betriebssystem, erkennt, um welchen Interrupt es sich handelt (bei mehreren möglichen) und ruft die Betriebssystem-Prozedur 'SIGNAL_INTERRUPT' auf. Als Parameter wird die Prozeß-Nr. des zu startenden Prozesses mitgegeben. Das Betriebssystem bringt mit Hilfe seiner Prozedur 'UNSUSPEND' den Prozeß in den Zustand 'Ablaufbereit', falls er sich im Zustand 'Warten auf Interrupt' befand. Vom PROCESS_SCHEDULER kann er daraufhin gestartet werden.

Dieses Verfahren kostet, bedingt durch den Prozeßwechsel, Zeit und bietet sich deshalb nur für Interruptbearbeitungen an, die längere Laufzeiten zur Folge haben (länger als der Prozeßwechsel). Das Betriebssystem wendet dieses Verfahren beim REAL_TIME_PROCESS an (Kapitel 2.8). Kurze Bearbeitungen macht man besser in einem speziellen Interrupt-Programm.

2.5.3 Semaphore, Kritische Region

Unter 'Kritischer Region' versteht man für alle Prozesse gemeinsame Betriebsmittel, die aber für die Dauer der Benutzung durch einen Prozeß vor dem Zugriff durch einen anderen Prozeß geschützt werden müssen (gegenseitiger Ausschluß, 'mutual exclusion'). Wird z.B. ein Prozeß während des Zugriffs auf eine kritische Region verdrängt, so darf trotzdem in dieser Zeit kein anderer auf die kritische Region zugreifen, um nicht Fehlerfälle, z.B. durch verändern von Daten, zu erzeugen. Bei der kritischen Region kann es sich z.B. um gemeinsame Datenbereiche handeln oder um externe Geräte wie Floppy-Disk-Speicher.

Wir wollen die Betrachtung der kritischen Region hier auf gemeinsame Datenbereiche beschränken.

Der Zugriff auf gemeinsame Bereiche wird vom Betriebssystem geregelt. Es handelt sich dabei um die sog. 'Semaphores'. Semaphore bedeutet soviel wie 'Ampel' und beinhaltet eine Kennzeichnung des kritischen Bereichs. Ist die Ampel auf rot, so ist der Bereich gesperrt, bei grüner Ampel ist der Bereich frei. Allgemein versteht man darunter einen Koordinator, der den gleichzeitigen Zugriff auf gemeinsam zugeteilte Einheiten, sowie die Prozeß-Synchronisation, regelt. Ein Prozeß, der die Zugriffserlaubnis auf kritische Daten vom Betriebssystem anfordert, ruft die entsprechende Betriebssystem-Prozedur (z.B.'REQUEST_REGION') auf. Als Parameter übergibt er den Semaphore, von dem er die Erlaubnis auf Zugriff erhalten möchte. Ist der Zugriff momentan nicht an einen anderen Prozeß vergeben, so kennzeichnet das Betriebssystem den Semaphore als an den anfordernden Prozeß vergeben. Nach Rückkehr aus der Prozedur darf der Prozeß darauf zugreifen. Falls der Zugriff bereits vergeben ist, wird der Prozeß vertagt. D.h., in dem Semaphore wird der Prozeß in eine Warteschlange aufgenommen und sein Zustand wird auf 'Warten auf Semaphore' gesetzt. Der

PROCESS_SCHEDULER startet einen anderen wartenden Prozeß. Benötigt ein Prozeß den ihm erteilten Zugriff nicht mehr, so hat er ihn z.B. mit 'RELEASE_REGION' an das Betriebssystem zurückzugeben. Das Betriebssystem kennzeichnet den Semaphore daraufhin als 'frei' und schaut in der Warteschlange nach, ob weitere Prozesse darauf warten. Ist dies der Fall, so teilt das Betriebssystem dem am längsten wartenden Prozeß (FIRST_IN_QUEUE) den Zugriff zu. Er wird in den Zustand 'ablaufbereit' gebracht und kann daraufhin vom PROCESS_SCHEDULER gestartet werden.

2.6 Prozeß-Synchronisation

In den vorangegangenen Kapiteln wurden die Betriebsmittel zur Prozeßkommunikation beschrieben. In diesem Kapitel wird nun erläutert nach welchen Kriterien das Betriebssystem bei der Vergabe des Prozessors an die einzelnen Prozesse vorgeht.

2.6.1 Prioritäten

Ein Kriterium für den Start eines Prozesses ist dessen Priorität. Der Anwender ordnet jedem Prozeß eine Priorität zu. So werden Überwachungsprozesse und Prozesse, die stark an einen Realtime-Ablauf gebunden sind, auf den höchsten Prioritätsstufen liegen (z.B. REAL_TIME_PROCESS). Dagegen können Prozesse, die diesen Echtzeitbezug nicht in so engen Grenzen haben, wie z.B. ein Prozeß zur Bedienung eines Bedienblattschreibers mit langsamer Baudrate, auf den niedrigeren Prioritäten liegen. Die Anzahl der Prioritätsstufen ist in den einzelnen Betriebssystemen unterschiedlich, im ROSY sind es z.B. 5 Stufen. Dabei stehen dem Anwender 4 Stufen zur Verfügung. Die niedrigste Priorität wird dem Idle-Prozesse zugeordnet. Der Idle-Prozeß ist der Prozeß, der gestartet wird, wenn kein Anwender-Prozeß ablauffähig ist. Er ist dem Betriebssystem zugeordnet und kann sich nicht vertagen. Damit hat er auch keine Möglichkeit von anderen Prozessen Nachrichten zu empfangen. Wird ein anderer Prozeß ablaufbereit, so wird der Idle-Prozeß sofort vom Betriebssystem verdrängt. In der Regel hat er keine Funktion, man kann aber in ihm z.B. einen Speichertest

im Hintergrund mitlaufen lassen oder einen 'Watchdog- Retrigger' durchführen (Kapitel 5.2.3).

In den meisten Betriebssystemen wird vom PROCESS_SCHEDULER der am längsten wartende Prozeß mit der höchsten Priorität zum Ablauf gebracht (Kapitel 2.6.4.2). Die ablaufbereiten Prozesse werden in den PROCESS_QUEUEs verkettet. Für jede Priorität ist eine PROCESS_QUEUE vorhanden. Abb. 2-27 zeigt als Beispiel die PROCESS_QUEUEs eines Betriebssystems mit 4 Prioritäten.

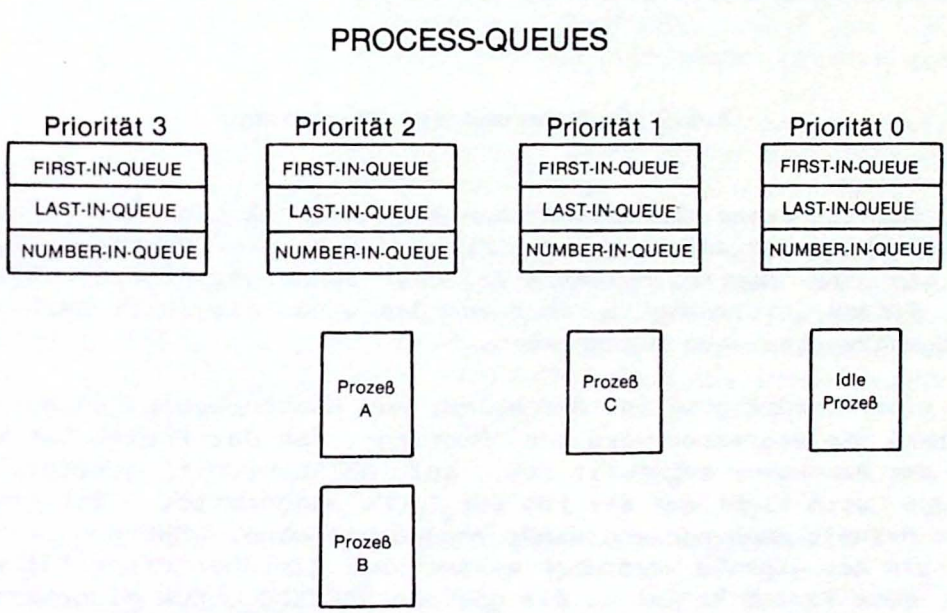

Abb. 2-27: PROZESS-QUEUES

Bei der höchsten Priorität (Priorität 3) wartet kein Prozeß auf den Prozessor. Mit der Priorität 2 warten die Prozesse A und B, mit der Priorität 1 der Prozeß C. Priorität 1 ist die niedrigste für vom Anwender eingebrachte Prozesse. Die nächstniedrigere (Priorität 0) ist dem Idle-Prozeß vorbehalten. Im PROCESS_SCHEDULER würde das Betriebssystem feststellen, daß die höchste Priorität, auf der Prozesse ablaufbereit sind, die Priorität 2 ist. Der am längsten wartende Prozeß in dieser PROCESS_QUEUE ist der erste, nämlich Prozeß A (FIFO-Prinzip). Ihm würde als erstes der Prozessor zugeteilt werden. Vorausgesetzt es kommen keine

weiteren Prozesse dazu, so läuft anschließend Prozeß B und danach Prozeß C. Hat sich auch dieser Prozeß vertagt, so ist in den PROCESS_QUEUES 1-3 kein Prozeß mehr eingekettet. Das Betriebssystem startet daraufhin den auf Priorität 0 wartenden Idle-Prozeß. Wird ein anderer Prozeß ablaufbereit, so wird der Idle-Prozeß sofort verdrängt. Veranlaßt wird dies z.B. durch einen Interrupt von einem externen Gerät oder durch den Ablauf eines Timers vom Betriebssystem.

Die Begriffe 'Verdrängung' und 'Vertagung'werden im folgenden nochmals erläutert (siehe auch Kapitel 2.1.1).

2.6.2 Veränderung von Prozessen

Der Ablauf eines Prozesses kann auf zwei Arten unterbrochen werden. Entweder auf Wunsch des Prozesses selbst (Vertagung, wird im nächsten Kapitel beschrieben) oder indem das Betriebssystem den Prozeß 'verdrängt'. Im folgenden wird ein üblicher Verdrängungsmechanismus beschrieben.

Bei einer Verdrängung ist der Ablauf nur unterbrochen, d.h. der Zustand des Prozesses wird von 'Running', den der Prozeß hat wenn ihm der Prozessor zugeteilt ist, auf 'Ablaufbereit' gesetzt. Der Prozeß wird nicht aus der PROCESS_QUEUE ausgekettet. Ein Prozeß kann deshalb auch nur von einem Prozeß mit einer höheren Priorität als der eigenen verdrängt werden. Bei gleicher Priorität wird der neue Prozeß hinten in die gleiche PROCESS_QUEUE eingekettet. Da bei der am häufigsten verwendeten Scheduling-Methode der am längsten wartende Prozeß läuft (erster in der Queue), kann keine Verdrängung stattfinden. Eine Verdrängung kann auf verschiedene Arten angestoßen werden. Die häufigsten sind:

- Ein Prozeß sendet an einen höher priorisierten Prozeß eine Nachricht. Das Betriebssystem bringt den Prozeß in der Betriebssystem-Prozedur 'UNSUSPEND' in den Zustand 'Ablaufbereit' und kettet ihn in die zur Priorität gehörende PROCESS_QUEUE ein. Der anschließend ablaufende PROCESS_SCHEDULER (Kapitel 2.6.4.2) stellt fest, daß ein höher priorisierter Prozeß ablaufbereit ist und teilt ihm den Prozessor zu.

- Für einen höher priorisierten Prozeß, der sich im Zustand 'Warten auf Interrupt' befindet, tritt der erwartete Interrupt auf. Die Prozedur SIGNAL_INTERRUPT macht ihn ablaufbereit, der anschließend laufende PROCESS_SCHEDULER startet ihn.
- Im Betriebssystem läuft ein Timer für einen höher priorisierten Prozeß ab. Das Betriebssystem versendet an den Prozeß eine Nachricht (siehe oben).
- Ein Prozeß gibt einen Semaphore zurück, auf den bereits ein höher priorisierter Prozeß wartet. Die aufgerufene Prozedur 'RELEASE_REGION_' kettet den Prozeß aus der Semaphore-Warteschlange aus, die Prozedur 'UNSUSPEND' kettet ihn in die PROCESS_QUEUE ein und der 'PROCESS_SCHEDULER' startet ihn.

Will man verhindern, daß Prozesse an bestimmten Stellen verdrängt werden, so kann man die Priorität durch Aufruf der Betriebssystem-Prozedur 'SET_PRIORITY' verändern (erhöhen). Nach Abarbeitung der kritischen Sequenzen ist durch Aufruf der Prozedur 'RESET_PRIORITY' der Prozeß wieder auf seine alte Priorität zu setzen. Bei Verwendung dieses Systemaufrufes hat der Anwender darauf zu achten, daß das Echtzeitverhalten des Systems durch zu häufiges und zu langes Verändern der Prioritäten nicht verloren geht (siehe Kapitel 6.2.3.2). Auch darf sich ein Prozeß nicht vor dem Rücksetzen der Priorität vertagen.

2.6.3 Vertagung, Fortsetzung von Prozessen

Die Vertagung eines Prozesses wird allein vom Anwender bestimmt und erfolgt somit an genau definierten Stellen innerhalb des Prozesses. Bei der Vertagung wird der Prozeß mit der Betriebssystem-Prozedur 'SUSPEND' aus der PROCESS_QUEUE ausgekettet. Der Zustand des Prozesses wird auf den Vertagungsgrund gesetzt. U.U. erfolgt ein Einketten in die, seinem Vertagungsgrund entsprechende Warteschlange. Die Fortsetzung des Prozesses erfolgt, wenn der Grund der Vertagung wegfällt. Der Prozeß wird anschließend wieder hinten in seine PROCESS_QUEUE eingekettet und kann dann gestartet werden. Die wichtigsten Gründe für eine Vertagung sind:

- Ein Prozeß wartet auf eine Nachricht. Stellt die von ihm aufgerufene Prozedur 'RECEIVE_EVENT' fest, daß sich momentan kein Event-Block in seiner EVENT_QUEUE befindet, so setzt

das Betriebssystem den Prozeßzustand auf 'Warten auf Nachricht' und kettet ihn aus der PROCESS_QUEUE aus.
Versendet ein anderer Prozeß (oder das Betriebssystem beim Ablauf eines Timers) eine Nachricht an den so vertagten, so versetzt das Betriebssystem den Prozeß, nach dem Einketten der Nachricht in die EVENT_QUEUE, in den Zustand 'Ablaufbereit' und kettet ihn in die PROCESS_QUEUE ein. Er kann daraufhin fortgesetzt werden.

- Ein Prozeß wartet auf einen Interrupt. Er wird in der von ihm aufgerufenen Prozedur 'WAIT_INTERRUPT' in den Zustand 'Warten auf Interrupt' gesetzt und aus der PROCESS_QUEUE ausgekettet. Der aufgetretene Interrupt macht dies mit Hilfe der Prozedur 'SIGNAL_INTERRUPT' wieder rückgängig.
- Ein Prozeß fordert einen Semaphore an, der nicht verfügbar ist. Sein neuer Zustand ist 'Warten auf Semaphore'. Die Rückgabe des Semaphores von einem anderen Prozeß, stößt die Fortsetzung an.
- Ein Prozeß kann sich auch ohne Angabe eines Grundes vertagen. Dazu ruft er die Prozedur 'DELAY' auf, sein Zustand wird auf 'Warten auf Fortsetzung' gesetzt.

  Die Fortsetzung des Prozesses wird von einem anderen Prozeß angestoßen, indem er die Prozedur 'CONTINUE' aufruft.

  Die meisten Realzeit-Betriebssysteme bieten Prozessen eine Möglichkeit sich für eine bestimmte Zeit zu vertagen, z.B. mit dem Aufruf 'SLEEP'. Das Betriebssystem weckt den Prozeß nach Ablauf der gewünschten Zeit auf. Im Betriebssystem ROSY (s. Anhang 1) ist diese Möglichkeit überflüssig, da dort eine komfortable Zeitverwaltung über 'Timer' (s. 2.7) geboten wird, die die 'SLEEP'-Funktion mit abdeckt.

2.6.4 Verwaltung der Prozeß-Synchronisation

Bei der Synchronisation der Prozesse hat der PROCESS_SCHEDULER prozeßspezifische Werte, wie Zustand und Priorität, zu bewerten. Es liegt deshalb nahe, alle den Prozeß bezeichnende Daten in einem zusammenhängenden Feld, dem PROCESS_CONTROLL_BLOCK (PCB), abzulegen.

2.6.4.1 Prozeß-Control-Block (PCB)

Abb. 2-28: Prozeß-Control-Block

Abb. 2-28 zeigt den Aufbau des PCBs, wie er im Betriebssystem ROSY im Anhang verwendet wird. Wie in den letzten Kapiteln beschrieben, werden in die PROCESS_QUEUEs die ablauffähigen Prozesse eingekettet. Dort wird nicht der Prozeß, also der Code, verkettet, sondern ein Datenfeld, das den Prozeß vertritt. Dieses Datenfeld ist der PCB. Im einzelnen enthält er:

- Den FORWARD_LINK, er ermöglicht die Verkettung des PCBs in den PROCESS_QUEUEs.
- Die beiden Stackadressen, SP und SS (für Prozessor 8086/8088). Sie werden bei der Prozeßinitialisierung (Kapitel 2.4.3) mit den für den Prozeß gültigen Adressen beschrieben. Bei jedem Prozeßwechsel ist darin der Inhalt der Register SP und SS abzulegen. Weitere Register sind nicht abzuspeichern. Diese beiden kennzeichnen allein die Stelle, an der der Prozeß vertagt oder verdrängt wurde (vgl. Kapitel 2.4.2 'Der Stack im Betriebssystem').
- Den Prozeßzustand. Die Prozedur 'SUSPEND' trägt ihn beim Vertagen ein (z.B. 'Warten auf Nachricht', 'Warten auf Interrupt'). Tritt ein Ereignis für diesen Prozeß ein, so prüft die Prozedur 'UNSUSPEND', ob sich der Prozeß in einem Wartezustand auf dieses Ereignis befindet, indem sie den Zustand aus dem PCB ausliest und vergleicht.
  Die Priorität des Prozesses. Beim Ein- bzw. Ausketten des PCBs in/aus der PROCESS_QUEUE benötigt das Betriebssystem die Information, welche PROCESS_QUEUE zu dem Prozeß gehört.

2.6.4.2 Prozeß_Scheduler

Die Aufgabe des PROCESS_SCHEDULER ist es, den Prozessen den Prozessor zuzuteilen bzw. zu entziehen. Wie bereits beschrieben wurde, müssen, um einen Prozeßwechsel zu vollziehen, nur die Stackregister umgeladen werden (Kapitel 2.4.2). Der Ablauf im PROCESS_SCHEDULER ist folgendermaßen:

Als erstes wird der Interrupt gesperrt (DISABLE). Beim Umladen der Stackadressregister darf keines dieser Register benutzt werden. Anschließend werden die Stackregister des gerade laufenden Prozesses in dessen PCB kopiert. Der PROCESS_SCHEDULER hat nun den Prozeß auszuwählen, der jetzt zum Ablauf kommen soll (es kann auch wieder der gleiche wie vorher sein, ein Prozeßwechsel findet dann nicht statt). Die Methode, nach der der Scheduler den Prozeß, der zum Ablauf kommen soll, auswählt, ist in den einzelnen Realzeit-Betriebssystemen unterschiedlich. Allen gemeinsam ist, daß immer die ablaufbereiten Prozesse mit der höchsten Priorität als erstes den Prozessor zugeteilt bekommen. Die Auswahl der PROCESS_QUEUE mit der höchsten Priorität, die ablaufbereite Prozesse enthält, erfolgt, indem der PROCESS_SCHEDULER die PROCESS_QUEUEs, beginnend bei der mit der höchsten Priorität, auf das Vorhandensein von eingeketteten PCBs untersucht (NUMBER_IN_QUEUE /= 0). Bei der Auswahl des Prozesses, der den Prozessor erhalten soll, aus denen die auf gleicher Prioritätsebene liegen (also in der gleichen PROCESS_QUEUE eingekettet sind), kommen die verschiedenen Methoden zur Anwendung.

Die wichtigsten Scheduling-Methoden sind:

- Eine Verdrängung zwischen Prozessen gleicher Priorität findet nicht statt. Dies ist die gebräuchlichste Scheduling-Methode. Ein anderer Prozeß auf gleicher Priorität erhält also erst dann den Prozessor, wenn der gerade laufende Prozeß sich vertagt. Um den Realzeitbezug zu erhalten, muß der Programmierer darauf achten, daß die Laufzeit seines Prozesses nicht zu lang wird.

- 'Round-Robin-Scheduling': Die Eingangswarteschlange eines Prozesses wird nicht abgearbeitet bis kein Poolelement mehr vorhanden ist. Es wird vielmehr nur ein einziges Poolelement pro Prozeß abgearbeitet und danach auf den nächsten Prozeß

gleicher Priorität umgeschaltet. Der Verdrängungsmechanismus bei Prozessen verschiedener Priorität bleibt derselbe.

- Timesharing: Den ablaufbereiten Prozessen gleicher Priorität wird eine feste Zeitscheibe zugeteilt, während ihnen der Prozessor zur Verfügung steht. Der Prozeßwechsel innerhalb einer Priorität findet also nach genau festgelegten Zeiten statt. Mit dieser Methode können keine strengen Realzeitbedingungen erfüllt werden, da die Prozeßverdrängung nicht aufgrund bestimmter Ereignisse (events) geschieht, die schnelle Reaktion erfordern, sondern wegen des Ablaufs einer Zeitscheibe. Die Methode wurde hier trotzdem angeführt, weil einige Betriebssysteme (z.B. ZRTS 8000 von ZILOG) dieses Scheduling-Verfahren wahlweise neben einer Realzeit-Scheduling-Methode anbieten.

Nach der Auswahl des Prozesses, der zum Ablauf kommen soll (wurde kein anderer Prozeß gefunden, so ist dies der Idle-Prozeß), kopiert der Scheduler die in dessen PCB abgelegten Stackadressen in die zugehörigen Register. Der Interrupt kann nun wieder freigegeben werden (ENABLE).

Durch die Beendigung der Prozedur (RET-Befehl) wird vom Stack des ausgewählten Prozesses die Adresse übernommen, an der der Prozeß fortgesetzt wird (vgl. Kapitel 2.4.2 Der Stack im Betriebssystem). Der Prozeßwechsel ist damit beendet.

2.7 Zeitverwaltung

In vielen Anwendungen hat man die Forderung zu erfüllen, daß Aktionen zeitgesteuert ablaufen müssen. Einfache Beispiele dafür sind Blinkrhythmen von LEDs, Laufgeschwindigkeit eines Cursors auf einem Bildschirm, Überwachung der Dauer von Tastendrücken, etc. Auch in der Einführung wurde am Beispiel des Telefonapparates auf diese Problematik hingewiesen.

In Kapitel 7.2.2.2 (Beispiel Sportwettkampf) werden weitere Beispiele von zeitgesteuerten Aktionen auftreten.

In Multitasking-Systemen müssen solche zeitgesteuerten Aktionen für jeden Prozeß möglich sein. Da die Prozesse (quasi-)parallel ablaufen, kann auch die Zeitzuweisung nicht durch eine einfache Zählschleife erfüllt werden. Bei Unterbrechung des Prozesses in der Zählschleife wäre die Zahl der Schleifendurchläufe dann kein Maß mehr für die verstrichene Zeit.

In einem Multitasking-System mit parallel ablaufenden Prozessen muß daher die Zeitverwaltung von zentraler Stelle, nämlich im Betriebssystem, koordiniert werden. In der Praxis hat sich gezeigt, daß oft mehrere parallel und unabhängig voneinander laufende Zeitaufträge in einem Prozeß nötig sind. Diese Möglichkeit wird im Betriebssystem ROSY in Form eines sog. 'REAL_TIME_PROCESS' geboten.

Im folgenden wird ausführlich auf die Problematik der Zeitverwaltung eingegangen. Das Vorgehen in der Praxis ist so, daß der Prozeß, der eine zeitgesteuerte Aktion durchführen möchte, dem Betriebssystem einen Auftrag gibt, ihm (den Prozeß) nach Ablauf der gewünschten Zeit eine Nachricht, die in Form einer Meldung in seiner Mailbox abgelegt wird, zukommen zu lassen. Das Betriebssystem stellt also den Prozessen das Betriebsmittel Zeit- oder Timer-Verwaltung zur Verfügung. Bei der Zuweisung eines Zeitauftrages für einen Prozeß von dem Betriebssystem spricht man vom 'Setzen eines Timers'.

2.7.1 Zeitglieder ('Timer')

Das Betriebssystem stellt dem Anwender die Prozedur 'TIMER_ASSIGN' zur Verfügung. Erfolgt im Prozeß dieser Prozeduraufruf, so wird vom Betriebssystem für diesen Prozeß ein Timer gesetzt. In einem Parameter beim Prozeduraufruf wird die Anzahl der Zeiteinheiten angegeben, nach deren Ablauf der Prozeß vom Betriebssystem informiert werden möchte.

Ein Beispiel soll die prinzipielle Funktionsweise eines Betriebssystem-Timers verdeutlichen:

Auf einer Prozessor-Baugruppe ist eine Hardware-Zeitgeberschaltung aufgebaut, die an den Prozessor in festen Zeitabständen, z.B. alle 20ms, einen Interrupt abgibt. In dem Interrupt-Pro-

gramm, das daraufhin abgearbeitet wird, wird die Bearbeitung der gesetzten Timer der einzelnen Prozesse angestoßen (näheres hierzu in Kapitel 2.8). Hat der Anwender beim Prozeduraufruf angegeben, daß er einen Timer mit z.B. 9 Zeiteinheiten gesetzt haben möchte, so informiert ihn das Betriebssystem nach Ablauf der 9 * 20ms = 180ms darüber.

In bestimmten Anwendungsfällen kommt es vor, daß ein Prozeß sich mehrere Timer gleichzeitig setzen möchte. Um dies zu ermöglichen, muß der Anwender beim Prozeduraufruf 'TIMER_ASSIGN' in einem 2. und 3. Parameter einen 'TIMER_TYPE' und einen 'USER_CODE' angeben. Hat der Prozeß z.B. mehrere Geräte zu verwalten, so könnte der TIMER-TYPE die Funktion des Zeitauftrags kennzeichnen. Damit können für die Geräte verschiedene Zeitaufträge unterschiedlicher Funktion gesetzt werden. In den USER_CODE kann man die Geräteart und die Gerätenummer (falls es mehrere Geräte der gleichen Art gibt) angeben.

Die Bedeutung von 'TIMER_TYPE' und 'USER_CODE' liegt allein beim Programmierer und muß je nach Anwendungsfall festgelegt werden. In der Nachricht, die der Prozeß vom Betriebssystem über den Ablauf seines Timers erhält, ist der 'TIMER_TYPE' und 'USER_CODE' enthalten, so daß der Anwender das Ereignis richtig zuordnen kann.

Hardware-Reaktionen werden oft zeitüberwacht. Das bedeutet folgendes: ein Gerät soll z.B. auf eine Einstellung hin innerhalb einer festgelegten Zeit eine Reaktion an das System zurückgeben. Bei einer Fehlfunktion kann diese Reaktion ausbleiben. Um zu verhindern, daß der Prozeß, der die Reaktion des Gerätes erwartet, nun ewig wartet, setzt er sich bei der Einstellung des Gerätes einen Timer mit der maximalen Zeit, in der die Reaktion des Gerätes erfolgt sein muß. Läuft der Timer ab, so wird z.B. an der Bedienkonsole ein Fehlerausdruck ausgegeben. Reagiert das Gerät korrekt, so entfällt die Notwendigkeit, daß das Betriebssystem den Prozeß über den Ablauf des von ihm gesetzten Timers informiert. Der Zeitauftrag muß in diesem Fall also vom Prozeß vor dem Ablauf der Zeit gelöscht werden können. Realzeit-Betriebssysteme stellen dafür den Prozeduraufruf 'TIMER_RELEASE' zur Verfügung. In 2 Parametern müssen wieder 'TIMER_TYPE' und 'USER_CODE' angegeben werden. Mit dem Prozeduraufruf löscht das Betriebssystem den für den aufrufenden Prozeß gesetzten und durch 'TIMER_TYPE' und 'USER_CODE' gekennzeichneten Timer aus seiner Timerverwaltung (Kapitel 2.7.2.3).

Oftmals wäre es praktisch, wenn man mit dem Aufruf 'TIMER_RELEASE' auch alle Timer mit dem gleichen 'USER_CODE' löschen könnte. Dies ist z.B. vorteilhaft, wenn für ein Gerät mehrere Timer gesetzt waren und das Gerät außer Betrieb genommen wird. Der Anwender spart sich dadurch unter Umständen eine eigene Timerverwaltung, d.h. er muß sich nicht merken, welche Timer gerade gesetzt sind. Dies ist eine wünschenswerte Ergänzung zu vielen Betriebssystemen. In der weiteren Beschreibung des Timers wird zunächst nicht weiter darauf eingegangen. Im Kapitel 3.1.5.3 wird eine solche Prozedur beschrieben.

2.7.2 Organisation der Zeitverwaltung

Bei einigen Betriebssystemen ist nur das Setzen einer genau festgelegten Anzahl von Timern pro Prozeß möglich. Z.B. kann im iAPX86/30 (s. 4.1) nur ein Timer pro Prozeß gesetzt werden ('SLEEP'-Funktionsaufruf). Bei anderen Betriebssystemen (z.B. im Betriebssystem ROSY im Anhang) ist es möglich, 'beliebig viele' Timer pro Prozeß zu setzen. Entsprechend ergibt sich eine unterschiedliche Organisation der Zeitverwaltung in den entsprechenden Betriebssystemen. In den folgenden beiden Kapiteln werden die beiden Methoden der Timerverwaltung beschrieben.

2.7.2.1 Feste Timerzuordnung

Ist einem Prozeß eine bestimmte (kleine) Anzahl von Timern fest zugeordnet, so ist für jeden Prozeß die entsprechende Anzahl Timerstrukturen vorhanden. (Abb. 2-29)

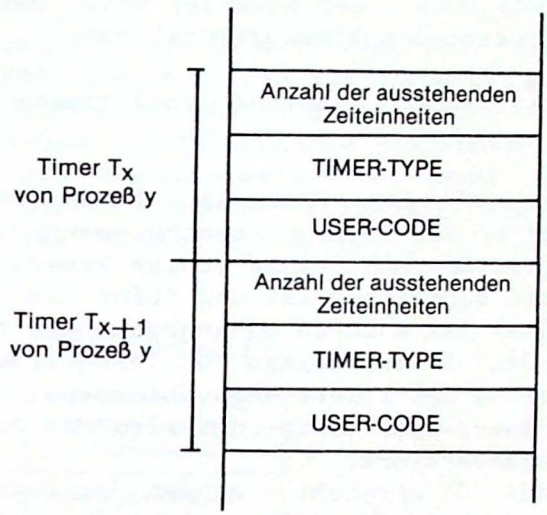

Abb. 2-29: Timerverwaltung bei fester Zuordnung zu den Prozessen

Jede Timerstruktur beinhaltet die 3 Variablen, die der Anwender beim Prozeduraufruf 'TIMER_ASSIGN' anzugeben hat:

- Die Anzahl der Zeiteinheiten: Das Betriebssystem 'merkt' sich hier, nach wievielen Zeiteinheiten der zugehörige Prozeß informiert werden möchte, d.h. wieviele Zeiteinheiten noch ausstehen, bis der Prozeß informiert werden muß. Ist die Anzahl der Zeiteinheiten '0', so ist dies für das Betriebssystem das Zeichen, daß dieser Timer nicht in Benutzung ist und erneut vergeben werden kann.

- Den TIMER_TYPE: Der Anwender hat damit die Möglichkeit, den Timer zu kennzeichnen. In den Nachrichten über den Ablauf des Timers wird der TIMER_TYPE dem Empfänger mitgeteilt.

- Den USER_CODE: Er dient zur weiteren Kennzeichnung des Timers für den Anwender. Wie auch beim TIMER_TYPE steht es dem Anwender völlig frei, wie er mit USER_CODE seinen Timer kennzeichnet.

Bei Betriebssystemen, die nur einen Timer pro Prozeß zulassen (z.B. iAPX86/30), entfällt die Kennzeichnung des Timers durch TIMER_TYPE und USER_CODE. Der Anwender weiß dann selbst, für welche Funktion er sich den Timer gesetzt hat.

Der Ablauf bei Zuweisung und Ablauf eines Timers ist folgendermaßen:

- Der Anwender ruft die Prozedur 'TIMER_ASSIGN' auf. Als Parameter gibt er die 3 vorher beschriebenen Größen an.
- Das Betriebssystem sucht einen freien Timer, der dem aufrufenden Prozeß zugeordnet ist und trägt die 3 Parameter ein. Ein freier Timer ist dadurch gekennzeichnet, daß in dem Timer als Anzahl der Zeiteinheiten '0' eingetragen ist. Damit ist die Zuweisung des Timers abgeschlossen.
- In jedem Hardware-Timer-Interrupt wird die Anzahl der Zeiteinheiten dekrementiert.
- Ist die Anzahl '0' erreicht, sendet das Betriebssystem dem Prozeß eine Nachricht (siehe 2.4.1.1). 'TIMER_TYPE' und 'USER_CODE' werden in der Nachricht mitgegeben. Damit ist der Timer wieder frei und kann erneut zugewiesen werden.

Ruft der Anwender vor Ablauf des Timers die Routine 'TIMER_RELEASE' auf, so setzt das Betriebssystem die Anzahl der Zeiteinheiten auf '0'. Im Hardware-Interrupt-Programm werden nur solche Timer behandelt, deren Anzahl nicht auf '0' steht. Damit findet im Interrupt-Programm kein Wechsel auf '0' statt und es wird keine Nachricht versendet.

2.7.2.2 Variable Timerzuordnung

In Betriebssystemen, bei denen jeder Prozeß 'beliebig viele' Timer setzen kann, kommt ein anderes Verfahren zur Anwendung:

Den einzelnen Prozessen sind die Timer nicht mehr fest zugeordnet. Stattdessen werden die notwendigen Informationen (TIMER_TYPE, USER_CODE, etc.) in entsprechend strukturierten Poolelementen abgelegt. Diese Poolelemente werden dann, sowohl in die pro Prozeß vorhandene 'TIMER_QUEUE', als auch in die im Betriebssystem einmal vorhandene 'TIME_SLOT_TABLE', eingekettet.

Man spricht hier von 'doppelt verketteten Listen'. Eine genaue Beschreibung dieses Verfahrens erfolgt in Kapitel 2.7.2.2.1 .

Die Zeitverwaltung erfolgt in der TIME_SLOT_TABLE (Abb. 2-30). Die TIME_SLOT_TABLE ist ein Feld aus Queue-Headern. Die Warteschlange, die an einem bestimmten Queue-Header hängt, besteht aus Poolelementen, in die die Zeitwünsche eingetragen sind. Im allgemeinen gehören die Poolelemente einer Queue zu verschiedenen Prozessen. Die Adressierung der einzelnen Queue-Header erfolgt über die Indizes des Feldes (Abb 2-30).

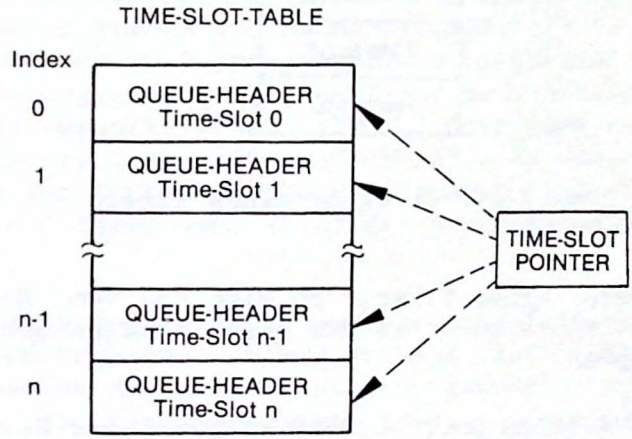

Abb. 2-30: TIME-SLOT-TABLE

Jeder möglichen Zeiteinheit ist ein Index zugeordnet. Die Zahl der vorhandenen Indizes (dargestellt durch den größten Index plus 1) ist somit gleich der maximalen Zahl der Zeiteinheiten, die vom Anwender angefordert werden können.

Es existiert eine Zeigervariable, der TIME_SLOT_POINTER, der zu einer bestimmten Zeit auf einen Time-Slot-Queue-Header zeigt. Dem TIME_SLOT_POINTER ist also zu jedem Zeitpunkt genau ein Index der TIME_SLOT_TABLE zugeordnet.

Bei jedem Zeitinterrupt, also z.B. alle 20 ms, wird der TIME_SLOT_POINTER um einen Index weitergeschaltet. Hat er den maximalen Index erreicht, wird er beim nächsten Zeitinterrupt wieder auf den Anfang gesetzt. Der TIME_SLOT_POINTER wird also beim

Zeitinterrupt inkrementiert modult den maximalen Index plus 1. Da so der TIME_SLOT_POINTER wie in einem Ring läuft, spricht man bei der TIME_SLOT_TABLE auch von einem 'Ringspeicher'.

Das Prinzip der Zeitverwaltung ist jetzt einfacher zu verstehen (Abb. 2-31):

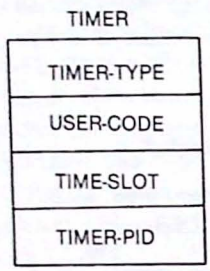

Abb. 2-31: Struktur eines Timers

Setzt ein Anwender einen Timer, so wird das vom Betriebssystem bereitgestellte Poolelement in die Queue desjenigen Time-Slots eingekettet, dessen

Time-Slot-Index = (TIME_SLOT_POINTER + Anzahl der Zeiteinheiten) MODULO Anzahl der Time-Slots.

beträgt. Beim Eintreffen des Zeitinterrupts brauchen dann nur noch alle Poolelemente ausgekettet und an die Zielprozesse versendet werden, die an den Queue-Header des aktuellen Index angehängt sind.

Hierzu 2 Beispiele:

Die Anzahl der Time-Slots ist 256, der Index läuft von 0 bis 255. Der TIME_SLOT_POINTER zeigt momentan auf den Time-Slot-Nr. 180. Der Hardware-Interrupt kommt alle 20ms.

1. Beispiel:

Der Anwender möchte einen Timer mit 460ms gesetzt haben. Sein Parameter 'Anzahl der Zeiteinheiten' beim Betriebssystem-Aufruf

'TIMER_ASSIGN' hat den Wert 460ms : 20ms = 23. Das Betriebssystem kettet das Timer-Poolelement in den Time-Slot

(180 + 23) MODULO 256 = 203

ein.

2. Beispiel:

Die Anzahl der gewünschten Zeiteinheiten beträgt 80 (≙1,6 s). Das Poolelement wird in den Time-Slot (180 + 80) MODULO 256 = 4 eingekettet.

Der Anwender veranlaßt durch den Prozeduraufruf 'TIMER_ASSIGN' das Setzen eines Timers für seinen Prozeß. Als Parameter übergibt er die Anzahl der Zeiteinheiten sowie TIMER-TYPE und USER-CODE, die es ihm ermöglichen, seinen Timer zu kennzeichnen. Das Betriebssystem vermerkt TIMER_TYPE und USER_CODE sowie den Time-Slot, in den der Timer eingekettet ist, in dem Poolelement, das den Timer in der TIME_SLOT_TABLE darstellt (Abb. 2-31). Außerdem wird noch die Prozeßnummer (PID) des aufrufenden Prozesses eingetragen.

Möchte der Anwender seinen Timer vor Ablauf löschen, so ruft er Betriebssystem-Prozedur 'TIMER_RELEASE' auf. Der Anwender hat keine Information darüber, in welchem Time-Slot sein Timer eingekettet ist, die Timerverwaltung liegt allein beim Betriebssystem. Die Time-Slot-Queues sind nicht nach Prozessen geordnet. In diesen Queues sind also Poolelemente von verschiedenen Prozessen gemischt, entsprechend dem Eintreffen der Zeitaufträge der Prozesse an das Betriebssystem.

Das Betriebssystem könnte nun alle Time-Slots durchsuchen, bis es den Timer, identifiziert durch PID, TIMER_TYPE und USER_CODE, gefunden hat. Dieser Suchvorgang kostet aber zuviel Zeit und kommt deshalb als Lösung nicht in Frage. Das Betriebssystem führt deshalb für jeden Prozeß eine 'TIMER_QUEUE'. In sie werden alle gesetzten Timer (Poolelemente) von dem zugehörigen Prozeß eingekettet. Es müssen also nur die Timer eines Prozesses durchsucht werden, was wesentlich günstiger ist. Erhält das Betriebssystem den Auftrag, einen Timer zu löschen, so findet es über die Timer_Queue die zu diesem Prozeß gehörenden Timer-Pool-Elemente. Hat es den betreffenden Timer identifiziert, so erhält es aus dem Poolelement die Information, in welchem Time-Slot der Timer ein-

gekettet ist, sowie welches Poolelement das vorherige und das nachfolgende ist (Kapitel 2.7.2.3).

Bei der Timerverwaltung ergibt sich somit die Notwendigkeit, die Timer-Poolelemente, sowohl in die Time-Slot-Table als auch in die Timer-Queue, einzuketten. Man spricht von doppelt verketten Listen.

2.7.2.2.1 Doppelt verkettete Listen

Abb. 2-32 zeigt den notwendigen Inhalt eines Poolelelments, das doppelt verkettet werden kann.

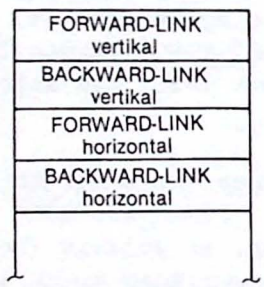

Abb. 2-32: Inhalt von doppelt verketteten Poolelementen

Die beiden Verkettungen werden hier als Vertikal bzw. Horizontalverkettung bezeichnet. Neben der Vorwärtsverkettung (FORWARD_LINK) wird auch das in der Queue voranstehende Poolelement geführt (BACKWARD_LINK). Der Vorteil liegt darin, daß beim Ausketten eines Poolelements (Timers) aus der Mitte einer Queue nicht die Queue nach der Adresse des in der Warteschlange voranstehenden Poolelements durchsucht werden muß (Kapitel 2.7.2.3).

Abb. 2-33 zeigt vier Poolelemente, die in zwei Queues eingekettet sind.

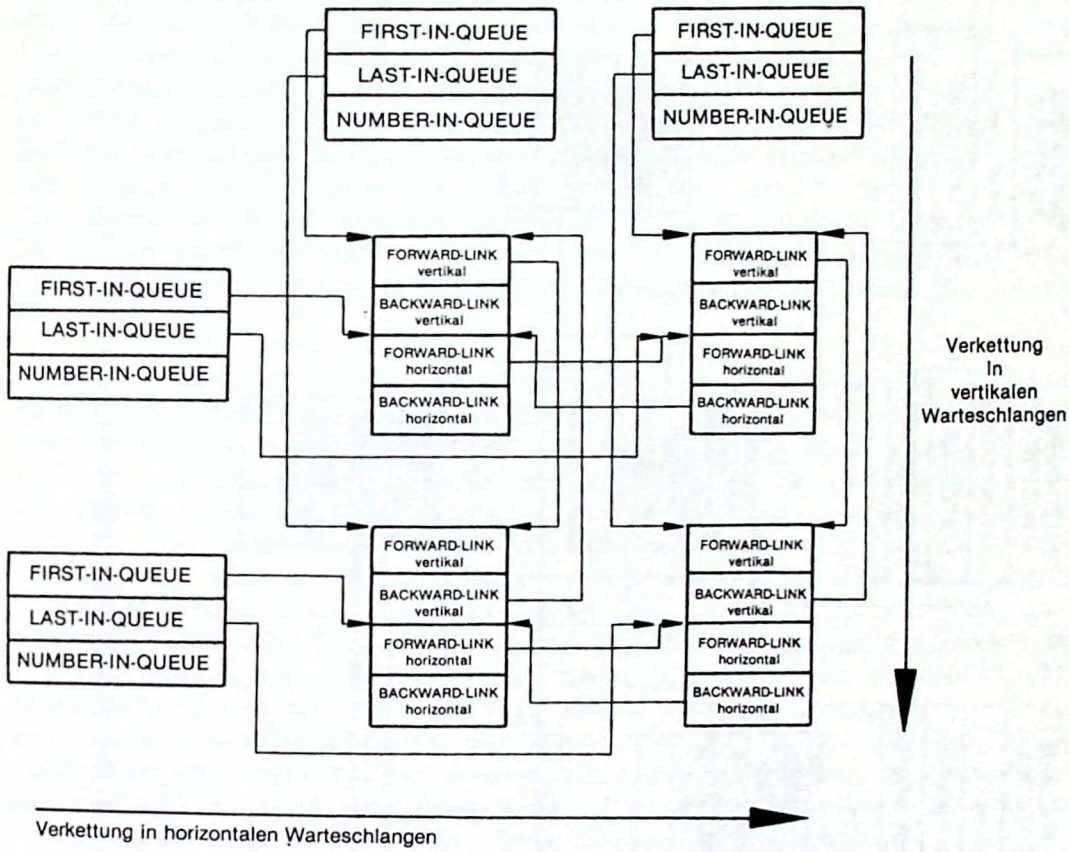

Abb. 2-33: Doppelt verkettete Poolelemente

Wie man sieht, erfolgt die Verkettung in einer Richtung (vertikal oder horizontal) immer innerhalb einer Queue. Die Poolelemente der Queue können aber in der anderen Richtung in unterschiedlichen Queues eingekettet sein.

2.7.2.3 Zuweisung und Löschen von Zeitgliedern

Anhand eines Beispiels über Zuweisung und Löschen von Zeitgliedern soll die Funktionsweise der doppelt verketteten Liste verdeutlicht werden.

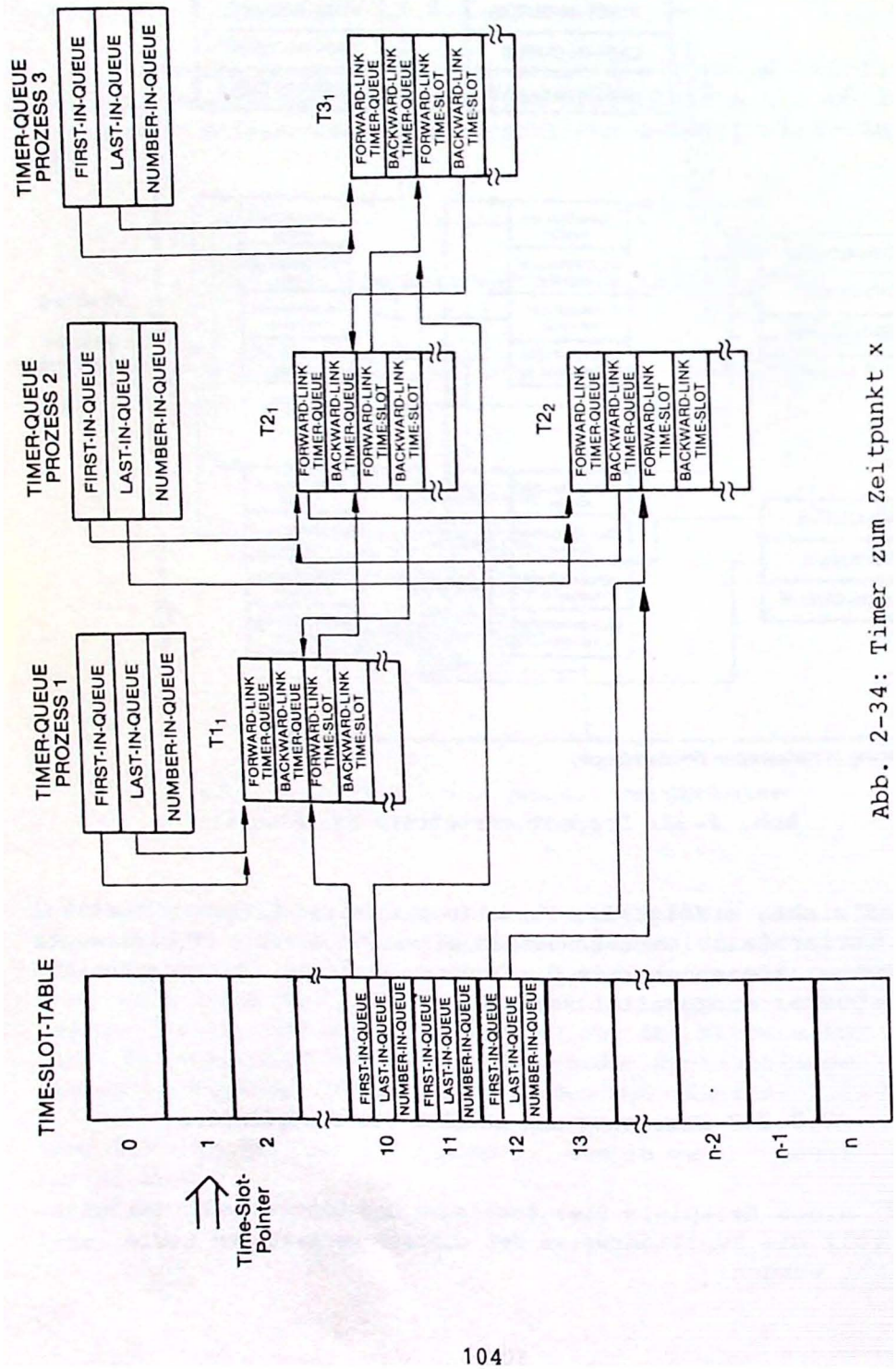

Abb. 2-34: Timer zum Zeitpunkt x

Abb. 2-34 zeigt den Zustand der TIMER_QUEUEs der Prozesse 1-3 und der TIME_SLOT_TABLE zu einem beliebigen Zeitpunkt x. Prozeß 1 hat einen Timer in Time-Slot 10 gesetzt (T11). Prozeß 2 hat 2 Timer gestartet, einen im Time-Slot 10 (T21) und einen im Time-Slot 12 (T22). Man sieht, daß über die Timer-Queue 2 die Timer T21 und T22 verkettet sind, da sie zum gleichen Prozeß gehören. Prozeß 3 hat ebenfalls einen Timer im Time-Slot 10 gesetzt. Über Time-Slot 10 sind deshalb die Timer T11, T21 und T31 verkettet. Der TIME_SLOT_POINTER zeigt auf den im nächsten Hardware-Interrupt zu bearbeitenden Time-Slot, in unserem Beispiel sei dies Time-Slot 1.

Prozeß 3 möchte nun einen Timer mit 11 Zeiteinheiten gesetzt haben. Das Betriebssystem kettet daraufhin ein weiteres Poolelement sowohl in die TIMER_QUEUE von Prozeß 3 als auch in den Time-Slot 12 ein (Abb. 2-35). Für die Einkettung ist im QHDR der TIMER_QUEUE 3 der Zeiger 'LAST_IN_QUEUE' auf das neue Poolelement T32 zu setzen. Der FORWARD_LINK_TQ vom Timer T31 ist mit der Adresse von T32 zu beschreiben. Damit ist T32 in die TIMER_QUEUE 3 eingekettet. Zur Einkettung in den Time-Slot 12 wird in den BACKWARD_LINK_TS die Adresse des bisher letzten Poolelements (T22) eingetragen. Sie steht im 'LAST_IN_QUEUE' von Time-Slot 12. Anschließend wird der 'LAST_IN_QUEUE' mit der Adresse von T32, dem neuen letzten Element der Queue, beschrieben. In den FORWARD_LINK_TS_ von T22 wird ebenfalls diese Adresse eingetragen. Nun ist T32 auch in den Time-Slot 12 eingekettet. Die Zuweisung eines neuen Timers durch das Betriebssystem ist beendet.

Möchte jetzt Prozeß 2 seinen Timer T21 gelöscht haben, so durchsucht das Betriebssystem als erstes die TIMER_QUEUE 2 nach dem zu löschenden Timer (TIMER_TYPE und USER_CODE müssen übereinstimmen). Im Beispiel ist es bereits beim ersten Timer (adressiert über FIRST_IN_QUEUE) erfolgreich. Das Ausketten aus der Queue erfolgt, indem in FIRST_IN_QUEUE der Inhalt vom FORWARD_LINK_TQ des auszukettenden Elements geschrieben wird. FIRST_IN_QUEUE zeigt anschließend auf T22. LAST_IN_QUEUE bleibt unverändert, T21 ist aus der Timer-Queue 2 ausgekettet. Zum Ausketten aus dem Time-Slot 12 ist der FORWARD_LINK_TS des in der Queue voranstehenden Timers (T11) auf die Adresse des nachfolgenden Timers (T31) zu setzen. T11 wird adressiert über den BACKWARD_LINK_TS, die Adresse von T31 steht im FORWARD_LINK_TS vom auszukettenden Timer T21. Abb. 2-35 zeigt die Verkettung der Timer nach dem Ausketten von Timer T21.

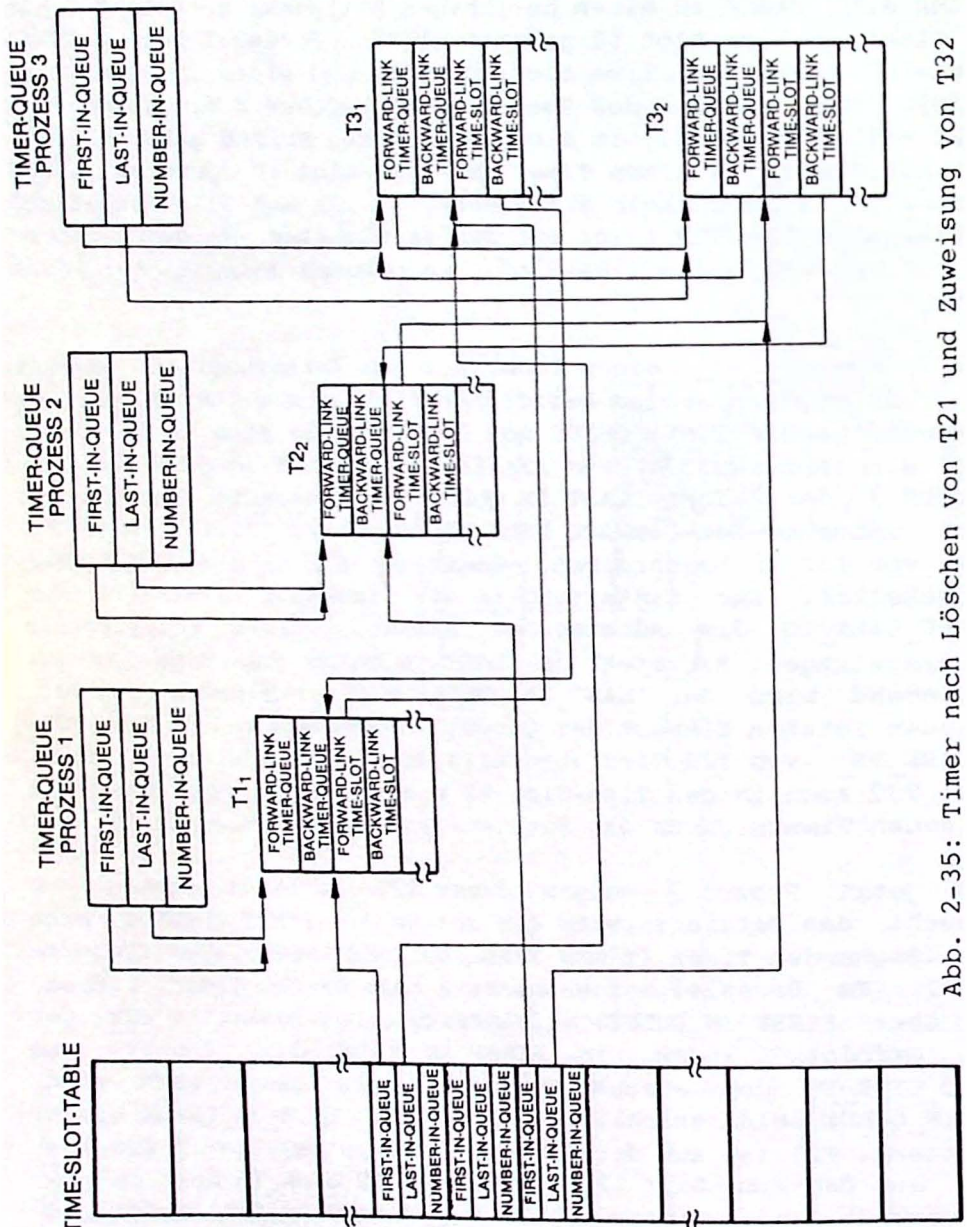

Abb. 2-35: Timer nach Löschen von T21 und Zuweisung von T32

Zum Abschluß sei noch bemerkt, daß man bei Timern, die vom Betriebssystem verwaltet werden, auch von sogenannten 'Software-Timern' spricht. Im Gegensatz dazu sind 'Hardware-Timer' integrierte Bausteine (z.B. Intel 8253), die nach Ablauf einer Zeit einen Impuls ausgeben. Dadurch wird hardwaremäßig eine Unterbrechung (Interrupt) erzeugt. In dem zugehörigen Programm wird dann die Verwaltung der 'Software-Timer' angestoßen.

2.7.2.4 Ereichbare Maximalzeiten

Möchte ein Prozeß einen Timer gesetzt haben, so ruft er die entsprechende Betriebssystem-Routine auf und übergibt die Anzahl der gewünschten Zeiteinheiten als Parameter. Bei den beiden unterschiedlichen Timerorganisationen ergeben sich verschiedene Maximalzeiten, auf die ein Timer gesetzt werden kann.

In Betriebssystemen, in denen die Timer den Prozessen fest zugeordnet sind, wird bei jedem Zeitinterrupt die als Parameter übergebene Anzahl der Zeiteinheiten dekrementiert. In der Regel ist der Parameter ein 16-Bit-Wort und kann die Werte von 0 bis 65535 annehmen. Die maximal mögliche Zeit errechnet sich aus den maximal möglichen Zeiteinheiten, multipliziert mit der Zeitbasis (z.B. 65535 * 20 ms = 1 310 700 ms = ca. 22 min.). Die Realisierung des Parameters als 8-Bit_Byte bringt, sowohl im Speicherbedarf wie auch in der Programmlaufzeit, keinen nennenswerten Vorteil.

Können die Timer den Prozessen frei zugeordnet werden, so ist die maximale Zeit zuerst einmal abhängig von der Anzahl der Time-Slots. Bei 256 Time-Slots (= ein Byte als Anzahl der Zeiteinheiten) und 20 ms Zeitbasis ist die maximal mögliche Zeit 256 * 20 ms = 5120 ms, also nur etwa 5 sec. In vielen Fällen reicht das nicht aus. Eine beliebige Erhöhung der Anzahl der Time-Slots ist aber oftmals nicht möglich, da nicht genügend Speicher zur Verfügung steht. Jeder Time-Slot beinhaltet einen Queue-Header, der z.B. im Betriebssystem ROSY 10 Byte Speicher belegt. D.h. allein die TIME_SLOT_TABLE mit 256 Time-Slots belegt über 2,5 kByte RAM-Speicher.

Eine einfache Methode, die Maximalzeit zu erhöhen, ohne mehr Speicher zu belegen ist, folgende:

Die Anzahl der gewünschten Zeiteinheiten wird vom Prozeß in einem 16-Bit-Wort als Parameter übergeben. Das Betriebssystem führt 256 Time-Slots. Das übergebene 16-Bit-Wort wird in 2 Byte aufgeteilt, das niederwertige Byte und das höherwertige Byte. Mit dem niederwertigen Byte wird der Time-Slot ausgerechnet, in den der Timer eingekettet wird (Kapitel 2.7.2.2). Das höherwertige Byte wird neben den anderen üblichen Informationen zusätzlich in dem Poolelement abgelegt, das den Timer in der TIME_SLOT_TABLE realisiert. Es gibt an, wie oft das Betriebssystem in seiner Timerbehandlung auf diesen Time-Slot treffen muß, bis der Timer abgelaufen ist. D.h. in der Timerbehandlung des Betriebssystems, die vom Timerinterrupt angestoßen wird, muß das Betriebssystem bei jedem Timer des aktuellen Time-Slots erst nachsehen, ob das höherwertige Byte auf '0' steht. Ist dies nicht der Fall, so muß dieses Byte dekrementiert werden und der Timer muß in der TIME_SLOT_TABLE eingekettet bleiben. Steht es auf '0', so ist der Timer abgelaufen und es sind die entsprechenden Aktionen auszuführen (Kapitel 2.8).

Diese Methode mit den Timern längere Zeiten zu erreichen, benötigt keinen zusätzlichen Speicherplatz (das zusätzlich benötigte Byte in dem Poolelement ist in der Regel vorhanden). Die Laufzeit der Timerbehandlung im Betriebssystem wird nur unwesentlich erhöht. Bei einer Zeitbasis von 20 ms lassen sich somit, wie bei der festen Timerzuordnung, fast 22 Minuten pro Zeitauftrag erreichen.

2.7.3 Genauigkeit des Timers

Die Ungenauigkeit der Zeiten liegt bei 'minus einer Zeiteinheit'. Das ergibt sich daraus, daß das Betriebssystem eben nur in seinen festen Zeiteinheiten, die vom Hardware-Zeit-Interrupt bestimmt werden, nachschaut, welcher Timer abgelaufen ist. (Abb. 2-36)

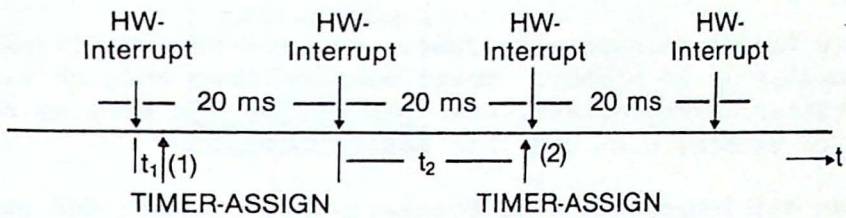

Abb. 2-36: Mögliche Fehler bei Zeitaufträgen

Die vom Anwender angegebenen Zeiteinheiten entsprechen der Anzahl der Hardware-Interrupts die erfolgen, bis das Betriebssystem den Aufrufer von dem Ablauf seines Timers informiert. Erfolgt der Prozeduraufruf 'TIMER_ASSIGN' kurz nach einem Hardware-Interrupt (Zeitpunkt (1) in Abb. 2-36), so beträgt die Ungenauigkeit nur die relativ kurze Zeit t1 vom letzten Interrupt bis zum Zeitpunkt des Prozeduraufrufs. Erfolgt der Aufruf dagegen kurz vor einem Interrupt (Zeitpunkt (2) in Abb. 2-36), so ist die Ungenauigkeit die Zeit t2 von fast 20ms. Daraus folgt, daß für kurze Zeit (TIMER_ASSIGN mit sehr wenigen Zeiteinheiten) der Fehler prozentual bis zu -100% betragen kann, während bei längeren Zeitaufträgen die Abweichung sehr klein ist.

Der mögliche Fehler berechnet sich:

F = -100 : Anzahl der Zeiteinheiten **Ä%Ü**

2.7.4 Echtzeit-Uhr

In Rechnersystemen, auf denen ein Realtime-Betriebssystem ablaufen soll, muß als Zeitbasis ein Hardware-Takt vorhanden sein. Er wird als Interrupt dem Prozessor zugeführt. In der zugehörigen Interrupt-Routine wird die Behandlung der Software-Timer, wie im vorherigen Kapitel beschrieben, angestoßen. Der Hardware-Takt ist in der Regel aus einem quarzstabilisierten Oszillator, z.B. dem Prozessortakt abgeleitet. Er weist somit eine ausreichende Genauigkeit auf, so daß es dem Betriebssystem möglich ist, softwaremäßig eine Uhr zu führen. Es handelt sich dabei um reine

Software-Zähler, die für die Stunden von 0-23, für Minuten und Sekunden von 0-59 und für Millisekunden von 0-999 laufen.

Kommt der Timerinterrupt alle 20ms, so wird der Millisekunden-Zähler jeweils um 20 erhöht. Tritt ein Übertrag auf, so wird der Sekundenzähler inkrementiert, usw. Das erstmalige Stellen der Uhr erfolgt von extern, z.B. über ein Bedienkommando.

Führt das Betriebssystem eine Echtzeit-Uhr, so hat der Anwender die Möglichkeit, Zeitaufträge in Absolutzeit zu erteilen. Ist die gewünschte Zeit erreicht, benachrichtigt das Betriebssystem den Anwender. Die Verwaltung der Absolutzeitaufträge erfolgt in einer CLOCK_QUEUE. Möchte der Anwender einen Zeitauftrag setzen, so ruft er die Betriebssystem-Prozedur 'SET_CLOCK' auf. Als Parameter gibt er die Absolutzeit in Stunden und Minuten sowie einen USER_CODE an. Der USER_CODE kennzeichnet die Funktion seines Zeitauftrags. Das Betriebssystem holt sich daraufhin ein Poolelement und schreibt die für seine Verwaltung notwendigen Werte hinein (Abb. 2-37).

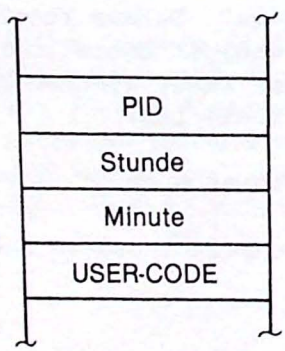

Abb. 2-37: Für die Echtzeituhr strukturiertes Poolelement

Neben den Parametern wird die Prozeß-Nr. (PID) abgespeichert. Das Betriebssystem kettet dieses Poolstück in die CLOCK_QUEUE ein. Um die Prozessorbelastung gering zu halten, sind in der CLOCK_QUEUE die Zeitaufträge in der Reihenfolge ihres Ablaufs eingekettet (Abb. 2-38).

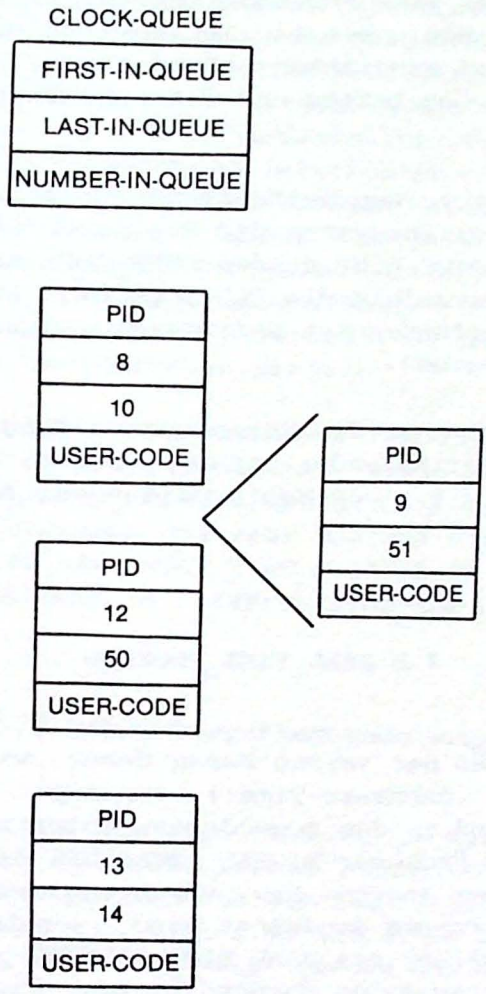

Abb. 2-38: Beispiel einer Clock-Queue

Sind z.B. die drei Aufträge 8.10 Uhr, 12.50 Uhr und 13.14 Uhr bereits in der Queue, so wird ein neuer Auftrag mit 9.51 Uhr zwischen 8.10 Uhr und 12.50 Uhr eingekettet. Damit ist die Zuweisung eines neuen Zeitauftrags abgeschlossen. Nach jedem Hardware-Interrupt wird der Millisekundenzähler um die Periodendauer des Hardware-Timers erhöht. Tritt ein Überlauf auf, wird der Sekundenzähler inkrementiert. Tritt auch hier ein Überlauf auf, wird der Minutenzähler erhöht. bei dessen Überlauf gilt für den Stundenzähler das gleiche. Bei jedem Überlauf des Millisekunden-

zählers, hat der REAL_TIME_PROCESS (Kapitel 2.8) die CLOCK_QUEUE solange zu durchsuchen, bis ein Zeitauftrag mit einem höheren Zeitwert als dem jetzt erreichten gefunden ist. Alle vorherigen Queue-Elemente sind auszuketten und die zugehörigen Prozesse sind zu benachrichtigen.

Möchte ein Anwender seinen Zeitauftrag vor Ablauf löschen, so stellt ihm das Betriebssystem die Routine 'CANCEL_CLOCK' zur Verfügung. Als Parameter gibt er den USER_CODE mit. Das Betriebssystem durchsucht daraufhin die CLOCK_QUEUE, bis das entsprechende Poolelement gefunden ist und kettet es aus. Es wird danach an den Pool zurückgegeben.

So ist natürlich nicht nur die Uhrzeitverwaltung möglich, sondern es kann ein kompletter Kalender geführt werden, was oft zu Dokumentationszwecken, z.B. von Fehlermeldungen, sehr wünschenswert ist.

2.8 REAL_TIME_PROZESS

Der REAL_TIME_PROCESS hat seinen Namen daher, weil er eng mit dem Realtime-Interrupt (Hardware-Timer) verknüpft ist. Mit jedem Timer-Interrupt wird in dem zugehörigen Interruptprogramm durch die Betriebssystem-Prozedur SIGNAL_INTERRUPT der REAL_TIME_PROCESS angestoßen. Der Anstoß- bzw. Vertagungsmechanismus für Prozesse, die an Interrupts angehängt sind, wurde in Kapitel 2.5.2 beschrieben. Die Aufgabe des REAL_TIME_PROCESS ist es, alle Funktionen des Betriebssystems durchzuführen, die mit dem festen Hardware-Zeittakt verbunden sind. Er wird deshalb als zum Betriebssystem gehörend betrachtet. Die Aufgaben sind im einzelnen:

- Abgelaufene Zeitaufträge feststellen und die Prozesse über den Ablauf unterrichten.
- Falls softwaremäßig eine Uhr geführt wird, aktualisieren der Uhrzeit.
- Falls Zeitaufträge in Echtzeit möglich sind, für abgelaufene Echtzeitaufträge die zugehörigen Prozesse verständigen.

Daß diese Funktionen in einem Prozeß abgehandelt werden und nicht in einem Interruptprogramm, hat den Vorteil, daß der Echtzeitbezug während der Prozeßbearbeitung bestehen bleibt. D.h. es können

weiterhin andere Interrupts bearbeitet werden. Würden die Funktionen in einem Interruptprogramm ausgeführt, so muß, während mit den Queues (Timer-Queue, Time-Slot-Table, etc.) gearbeitet wird, der Interrupt gesperrt werden. Ansonsten könnte das Timer-Interruptprogramm von einem Hardware-Interrupt unterbrochen werden. Würde in dem zugehörigen Interruptprogramm ein Prozeß angestoßen werden, dessen Aktionen Veränderungen an der gleichen, in Bearbeitung befindlichen Queue zur Folge hätten (z.B. TIMER_ASSIGN), so würde die Queue zerstört werden und das Betriebssystem wäre nicht mehr lebensfähig. Der REAL_TIME_PROCESS hat in der Regel die höchste Priorität, er kann deshalb nicht von anderen Prozessen unterbrochen werden.

Der Nachteil des REAL_TIME_PROCESS ist, daß durch den Start des Prozesses aus dem Interruptprogramm, sowie bei dem Prozeßwechsel nach dessen Beendigung eine Grundlast auf dem Rechner erzeugt wird. In kleinen Systemen mit sehr wenigen Timern kann es deshalb sinnvoll sein, auf den REAL_TIME_PROCESS zu verzichten und die Timerbearbeitung direkt im Interruptprogramm vorzunehmen.

2.9 Betriebssystemüberwachungen

Das Betriebssystem hat die Möglichkeit, in einigen seiner Routinen festzustellen, ob es weiterhin lebensfähig ist. Wird ein Fehler festgestellt, so kann es den Fehler allerdings nicht mehr beheben. Die Fehlerart kann z.B. in einem Tracebereich abgespeichert oder über LEDs angezeigt werden und der Prozessor wird anschließend gezielt ausfallen (HALT-Befehl). Der Tracebereich kann vom Anwender ausgelesen werden, sofern ihm ein geeignetes Mittel zur Verfügung steht (Testsystem).

Es können z.B. folgende Fehlerfälle erkannt werden:

- Bei der Vergabe von Poolelementen ist das System nicht mehr lebensfähig, wenn ein Poolelement angefordert wird, der Pool aber leer ist.
- Bei der Rückgabe von Poolelementen wird erkannt, daß ein Poolelement zurückgegeben wird, das dem aufrufenden Prozeß nicht gehört (OWNER_PID ungleich eigener Prozeßnummer). Das beinhaltet auch die Rückgabe eines Poolelements, das bereits zurückgegeben ist. Bei der Vergabe der Poolelemente und dem

Versenden von Nachrichten an einen anderen Prozeß, muß das Betriebssystem die Prozeßnummer im Poolelement aktualisieren.
- Im PROCESS_SCHEDULER können Stackfehler erkannt werden. Hat der Prozeß bei der Vertagung nicht mehr 'genug' Stack (z.B. für mindestens einen Interrupt und zwei Prozeduraufrufe), so liegt in der Regel ein Fehlerfall vor.
- Möchte ein Prozeß einen Timer löschen, der nicht mehr läuft und es steht auch keine Nachricht über den Ablauf in der EVENT_QUEUE, so handelt es sich um einen Programmierfehler.

2.10 Zusammenfassung

Anschließend sind noch einmal die wichtigsten Begriffe des Kapitels 2 zusammengefaßt.

- Prozeßkonzept: Aufteilung des Gesamtprogramms in einzelne, (quasi) parallel ablaufende Prozesse. Das geordnete Zusammenwirken der Prozesse wird vom Betriebssystem gesteuert.
- Warteschlangen: Hilfsmittel zur Zwischenspeicherung von Informationen, bei der die zeitliche Reihenfolge im allgemeinen erhalten bleibt.
- Poolverwaltung: Speicherplatz wird dem Anwender auf Anforderung zur Verfügung gestellt.
- Stackverwaltung: Jeder Prozeß erhält seinen eigenen Stackbereich. Dies ist die Grundlage für die Realisierung des Prozeßkonzepts.
- Prozeß-Kommunikation: Die Prozesse können untereinander Nachrichten austauschen, die zum Prozeßwechsel führen können.
- Prozeß-Synchronisation: Die Prozesse können in verschiedene Prioritätsebenen gelegt werden. Wird ein höher priorisierter Prozeß ablaufbereit, so werden Prozesse mit niedrigerer Priorität verdrängt. Warten Prozesse auf ein bestimmtes Ereignis oder einen bestimmten Zugriff, so werden sie vertagt.
- Scheduling: Es läuft der ablaufbereite Prozeß mit der höchsten Priorität. Sind mehrere Prozesse der gleichen Priorität ablaufbereit, so kommen verschiedene Methoden zur Anwendung.
- Zeitverwaltung: Die Prozesse können dem Betriebssystem den Auftrag geben, ihnen nach Ablauf einer von Ihnen festgelegten Zeit, eine Nachricht zukommen zu lassen.

- REAL_TIME_PROCESS: Er ist dem Betriebssystem zugeordnet und wird mit jedem Hardware-Zeitinterrupt gestartet. Er bearbeitet die gesetzten Timer.
- Betriebssystem-Überwachung: In verschiedenen Betriebssystem-Routinen kann die weitere Funktionsfähigkeit des Betriebssystems überprüft werden. Ist sie nicht mehr gegeben, erfolgt eine Fehlerbehandlung.

Zweiter Teil:

Praktische Betriebssysteme

Zweiter Teil

3. Anwendung auf einen exemplarischen Betriebssystem-Kern

3.1 Typische Benutzerschnittstelle eines Betriebssystem-Kerns

Die Benutzerschnittstellen zwischen Anwender und Betriebssystem bilden die Betriebssystem-Prozeduren. Sie werden vom Anwender aufgerufen und mit den nötigen Parametern versorgt. Im folgenden wird der Ablauf innerhalb der wichtigsten Prozeduren des Betriebssystem-Kerns sowie des REAL_TIME_PROCESS, der als zum Betriebssytem gehörend angesehen wird, beschrieben. Die Darstellung des Ablaufs erfolgt in Struktogrammen, deren Elemente als bekannt vorausgesetzt werden.

Der Ablauf ist im Detail in jedem Betriebssystem etwas unterschiedlich, auch bei der Namensgebung gibt es keine einheitliche Lösung. Die nachfolgende Beschreibung ist deshalb in diesen Punkten als beispielhaft anzusehen.

Im Anhang ist der Code des Multitasking-Realtime-Betriebssystems 'ROSY' (Realtime Operating System) und des zugehörigen REAL_TIME_PROCESS abgedruckt. Es handelt sich dabei um ein in PL/M86 geschriebenes Betriebssystem, das nach den hier beschriebenen Struktogrammen codiert wurde. Die Beschreibung der Leistungsmerkmale von ROSY erfolgt in Kapitel 3.4 .

3.1.1 Warteschlangenverarbeitung

Das Grundprinzip eines Betriebssystems sind die verketteten Listen. Zum Ein- und Ausketten werden eine Reihe von Prozeduren zur Verfügung gestellt. Im Normalfall wird an das Ende einer Queue eingekettet und vom Anfang ausgekettet (FIFO-Prinzip). Das Betriebssystem stellt dafür die Prozeduren 'QUEUE' und 'UNQUEUE' zur Verfügung. In einigen Spezialfällen ist es nötig, Elemente in der Mitte einer Queue ein- bzw. auszuketten. Die Prozeduren 'ENQUEUE' bzw. 'DEQUEUE' besorgen dies.

Das Betriebssystem verwendet diese Prozeduren zur Bearbeitung der von ihm verwalteten Queues (EVENT_QUEUE, PROCESS_QUEUE, etc.). Dem Anwender stehen sie zur Verfügung, um in seinen eigenen

Warteschlangen Verkettungen ausführen zu können. An dem vom Betriebssystem verwalteten Queues darf er damit nichts verändern. Abb. 3-1 zeigt die EVENT_QUEUE eines Prozesses, in die momentan die 3 Poolelemente A, B und C eingekettet sind. Anhand dieses Bildes kann der Ablauf in den einzelnen Prozeduren verfolgt werden.

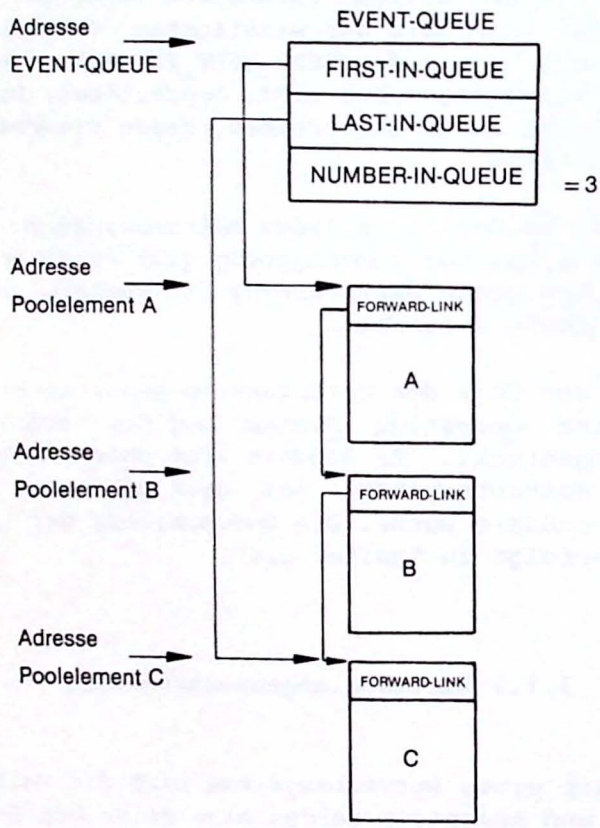

Abb. 3-1: Event-Queue mit drei Elementen

3.1.1.1 QUEUE, UNQUEUE

Soll ein neues Element an das Ende einer Queue angehängt werden, so ist die Prozedur 'QUEUE' aufzurufen. Als Parameter sind die Adressen des Queue-Headers (alle Queue-Header haben die gleiche Struktur!) sowie die Adresse des neu einzukettenden Elementes zu übergeben. Im folgenden Struktogramm werden sie 'QHDR_ADR' und 'NEW_ADR' genannt. Ein unterschiedlicher Ablauf ergibt sich je nachdem ob die Queue vor dem Einketten leer ist oder nicht (Abb. 3-2).

In den FORWARD_LINK des neuen (letzten) Queueelements ist als Zeichen, daß kein weiteres Element angehängt ist, die symbolische Adresse 'NULL' (z.B. FFFFH:FFFFH) einzutragen.

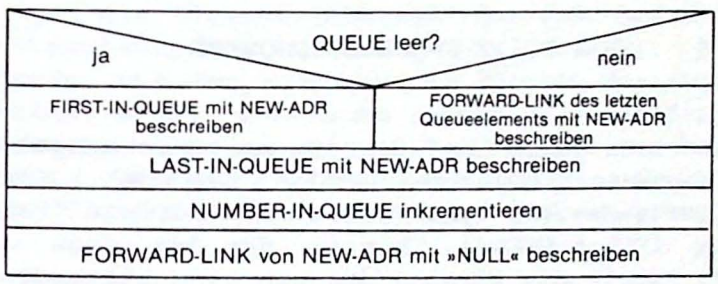

Abb. 3-2: Betriebssystem-Prozédur 'QUEUE'

Von der Prozedur 'UNQUEUE' wird die Adresse des ersten Elements einer Queue dem Aufrufenden übergeben. Parameter sind die Adresse des Queue-Headers (QHDR_ADR) und die Adresse eines Zeigers (ZEIGER_ADR), auf den die Adresse des ausgeketteten Elements (QUE_ELE_ADR) geschrieben werden soll. Der Speicherplatz, dessen Adresse sich in ZEIGER_ADR befindet, wird ZEIGER genannt. Befindet sich kein Element in der Queue, so gibt die Prozedur die symbolische Adresse 'NULL' zurück. Der Aufrufer muß die Gültigkeit der Adresse abfragen. Enthält die Queue nur ein Element, so ist neben dem 'LAST_IN_QUEUE' auch der 'FIRST_IN_QUEUE' zu beschreiben (Abb. 3-3).

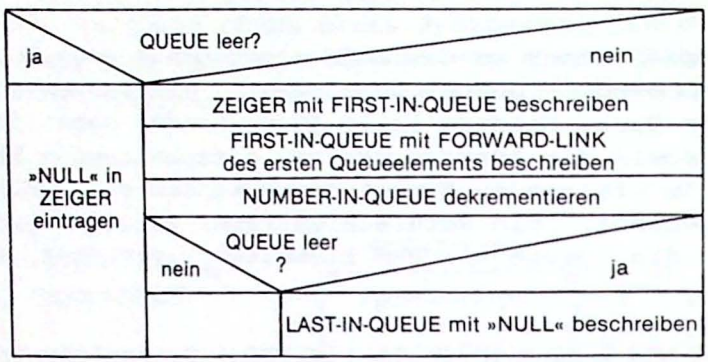

Abb. 3-3: Betriebssystem-Prozedur 'UNQUEUE'

3.1.1.2 ENQUEUE, DEQUEUE

Soll zwischen zwei Queueelementen ein weiteres eingekettet werden, so ist die Betriebssystem-Prozedur 'ENQUEUE' aufzurufen. Die Adresse des Queue-Headers (QHDR_ADR), die Adresse (PRE_ADR) des Queueelements (PRE_ELEMENT), hinter dem das neue eingekettet werden soll, sowie die Adresse des neu einzukettenden Elements (NEW_ADR), sind als Parameter zu übergeben.

Die Prozedur ermöglicht auch ein Einketten an erster bzw. letzter Stelle einer Queue. Der Anwender müßte sonst in Programmen, die auch ein Einketten in die Mitte einer Queue erfordern, zwischen den 3 Prozeduren ENQUEUE (in die Mitte einketten), QUEUE (an das Ende einketten) und REQUEUE (an den Anfang einketten, wird hier nicht beschrieben) wählen. Soll ein neues Element an den Anfang einer Queue eingekettet werden, so ist anstelle der Adresse des vorherigen Elements, die Adresse des Queue-Headers anzugeben. FIRST_IN_QUEUE wird dann wie der FORWARD_LINK behandelt (Abb. 3-4).

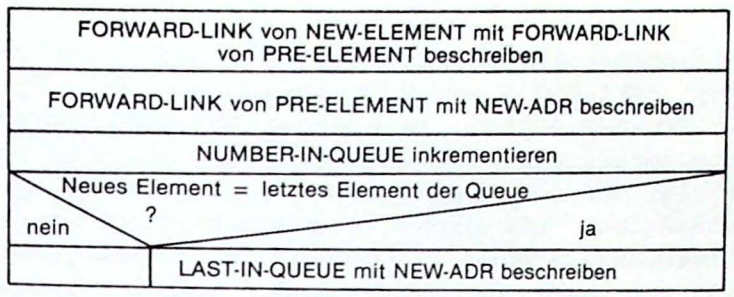

Abb. 3-4: Betriebssystem-Prozedur 'ENQUEUE'

Das Ausketten aus der Mitte einer Queue besorgt die Prozedur 'DEQUEUE'. Parameter sind die Adresse des Queue-Headers (QHDR_ADR) und die Adresse (PRE_ADR) des dem auszukettenden, in der Queue voranstehenden, Elements (PRE_ELEMENT). Bei Anwendung dieser Prozedur muß dem Anwender die Adresse des auszukettenden Elements bekannt sein, ansonsten könnte er nicht die Adresse des voranstehenden Elements angeben. Eine Übergabe durch die Prozedur entfällt somit. Ist das erste Element der Queue auszuketten, so ist auch hier die Adresse des Queue-Headers anzugeben (Abb. 3-5).

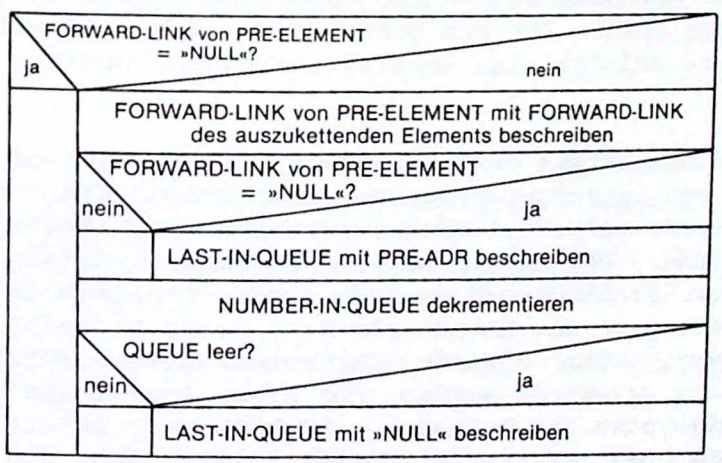

Abb. 3-5: Betriebssystem-Prozedur 'DEQUEUE'

3.1.2 Vergabe von Poolelementen

Benötigt ein Anwender ein Poolelement, so fordert er dies mit dem Prozeduraufruf 'GET_POOL' vom Betriebssystem an. Die Rückgabe erfolgt mit 'RELEASE_POOL'. In einigen Betriebssystemen werden mehrere Pools verwaltet. Sie unterscheiden sich in ihrer Länge, d.h. die Anzahl der dem Anwender zur Verfügung stehenden Bytes ist unterschiedlich. Die nicht vergebenen Poolelemente sind in den meisten Betriebssystemen in Queues aneinandergekettet. Jede Queue besteht wie immer aus Queue-Header und den Queueelementen. Bei der Anforderung bzw. Rückgabe wird dem Betriebssystem durch Angabe der Adresse des Queue-Headers gesagt, welcher Pool gewünscht ist.

3.1.2.1 GET_POOL

Benötigt ein Anwender ein Poolelement, so ruft er die Betriebssystem-Prozedur 'GET_POOL' auf. Beim Aufruf hat er zwei Parameter zu übergeben. Der erste ist die Adresse des Queue-Headers des gewünschten Pools (POOL_ADR). Beim zweiten handelt es sich um die Adresse (POOL_PTR_ADR) eines Speicherplatzes , auf den das Betriebssystem die Adresse des zur Verfügung gestellten Poolelements ablegen soll. Ist die Queue leer, d.h. kein Poolelement verfügbar, so erfolgt eine spezielle Fehlerbehandlung (Prozedur 'ERROR').

Während des Auskettens darf kein Interrupt den Ablauf unterbrechen. Würde das Interrupt-Programm einen Prozeß starten, der auch ein Poolelement anfordert oder zurückgibt, so könnte die Queue zerstört werden. Der Interrupt wird deshalb am Anfang der Prozedur gesperrt (DISABLE) und am Ende wieder freigegeben (ENABLE). Wegen der Freigabe des Interrupts darf diese Prozedur in Interruptprogrammen, die vollständig interrupt-geschützt ablaufen sollen, nicht verwendet werden. Für diese Anwendungsfälle stellt das Betriebssystem die Prozedur 'GET_POOL_DIS' bereit, bei der die Freigabe des Interrupts am Ende fehlt. Abb. 3-6 zeigt den Ablauf.

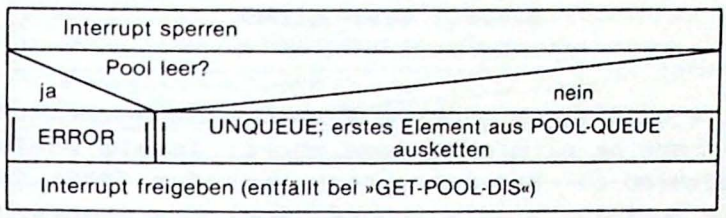

Abb. 3-6: Betriebssystem-Prozedur 'GET-POOL'

3.1.2.2 RELEASE_POOL

Mit der Prozedur 'RELEASE_POOL' wird ein nicht mehr benötigtes Poolelement an den Pool zurückgegeben. Als Parameter ist die Adresse des Queue-Headers des Pools (POOL_ADR) anzugeben, an den die Rückgabe erfolgen soll. Der zweite Parameter ist die Adresse des Poolelements (POOL_ELEMENT_ADR), das zurückgegeben wird. Das Betriebssystem kettet dieses Element hinten an die Pool-Queue an. Auch hier darf kein Interrupt den Ablauf stören (Abb. 3-7). Für Interruptprogramme wird die Prozedur 'RELEASE_POOL_DIS' bereitgestellt.

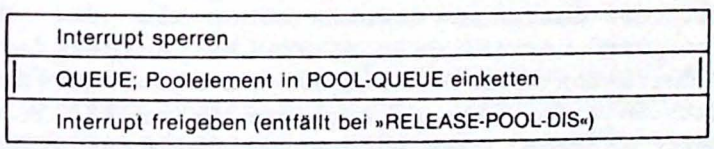

Abb. 3-7: Betriebssystem-Prozedur 'RELEASE-POOL'

3.1.3 Prozeß-Kommunikation

3.1.3.1 SEND_EVENT

Möchte ein Prozeß einem anderen Prozeß eine Nachricht zukommen lassen, so schreibt er die Meldung zuerst in ein Poolelement und ruft anschließend die Betriebssystem-Prozedur 'SEND_EVENT' auf. Als Parameter übergibt er die Prozeßnummer des Empfangsprozesses (REC_PID), sowie die Adresse des Poolelements, das seine Nachricht enthält (EVENT_PTR). Das Betriebssystem kettet daraufhin das Poolelement in die EVENT_QUEUE des Empfangsprozesses ein (Abb. 3-8). Während des Einkettens muß der Interrupt gesperrt sein. Anschließend prüft es, ob der Empfangsprozeß bereits in die PROCESS_QUEUE eingekettet ist (Prozeßzustand 'Ablaufbereit'). Ist dies der Fall, so kann der Prozeß vom PROCESS_SCHEDULER gestartet werden. Ansonsten ruft das Betriebssystem die Routine 'UNSUSPEND' auf.

Die Prozedur UNSUSPEND steht dem Anwender nicht zur Verfügung, sie wird nur vom Betriebssystem selbst verwendet. Ihre Aufgabe ist es, einen Prozeß ablaufbereit zu machen, sofern der Prozeß sich in einem als Parameter übergebenen Zustand, befindet. Als weiterer Parameter wird neben dem Zustand die Prozeßnummer des zu behandelnden Prozesses übergeben.

Beim Aufruf aus der Prozedur SEND_EVENT ist der übergebene Zustand 'Warten auf Nachricht'. Stimmt er mit dem Prozeßzustand überein, so wird der Prozeß in die PROCESS_QUEUE eingekettet. Ist die Priorität des Empfangsprozesses höher als die des gerade laufenden, so wird der 'PROCESS_SCHEDULER' aufgerufen. Er wird den momentan laufenden Prozeß vertagen und stattdessen, je nach Scheduling-Methode, den mit der höchsten Priorität z.B. am längsten wartenden, starten. Der Aufruf des 'PROCESS_SCHEDULER' erfolgt in Betriebssystemen bei denen immer der mit der höchsten Priorität am längsten wartende Prozeß den Prozessor erhält, gegebenenfalls nur im 'UNSUSPEND', da, falls der Prozeß sich bereits in der PROCESS_QUEUE befindet, er bereits laufen würde, wenn er eine höhere Priortät hätte (Abb. 3-9).

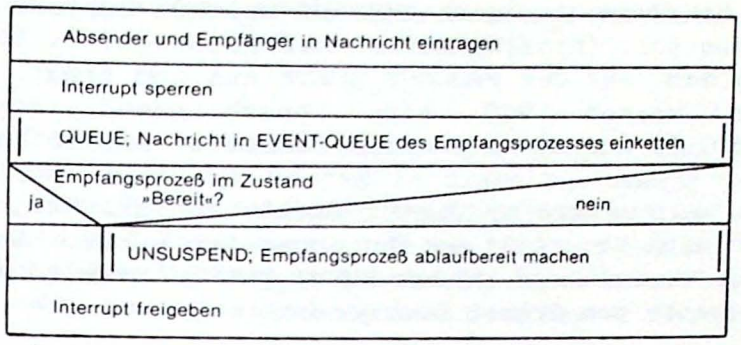

Abb. 3-8: Betriebssystem-Prozedur 'SEND-EVENT'

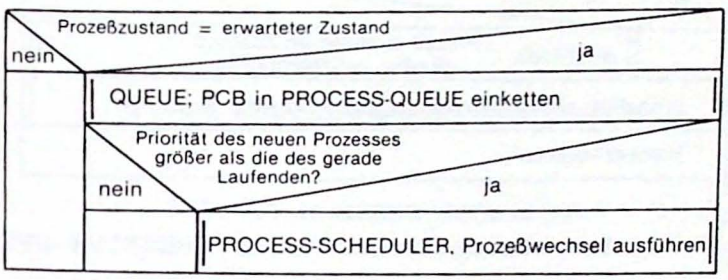

Abb. 3-9: Betriebssystem-Prozedur 'UNSUSPEND'

3.1.3.2 RECEIVE_EVENT

Um eine Nachricht aus seiner EVENT_QUEUE zu empfangen, ruft ein Prozeß die Betriebssystem-Prozedur 'RECEIVE_EVENT' auf. Als Parameter gibt er die Adresse eines Zeigers an (EVENT_PTR_ADR), auf den das Betriebssystem die Adresse der empfangenen Nachricht (EVENT_PTR) ablegen soll. Der Interrupt muß während des ganzen Prozedurablaufes gesperrt sein, damit die Queues nicht verändert werden können. Befindet sich ein Poolelement in der EVENT_QUEUE, so wird es ausgekettet und seine Adresse dem Prozeß übergeben (Abb. 3-10). Ist kein Poolelement in der EVENT_QUEUE, so wird der Prozeß vertagt. D.h., es wird die Prozedur 'SUSPEND' aufgerufen. SUSPEND ist das Gegenstück zu UNSUSPEND und vertagt den gerade laufenden Prozeß. Sie steht ebenfalls nur dem Betriebssystem zur

Verfügung. Ein Parameter gibt an, auf welchen Zustand der Prozeß gesetzt werden soll (hier: 'Warten auf Nachricht'). SUSPEND kettet den Prozeß aus der PROCESS_QUEUE aus und trägt den neuen Zustand in seinen PCB ein. Anschließend startet der PROCESS_SCHEDULER den nächsten ablaufbereiten Prozeß (Abb. 3-11).

Der Prozeß wird später an dieser Stelle fortgesetzt, wenn mit 'SEND_EVENT' eine Nachricht an ihn abgegeben wurde. Es befindet sich dann ein Poolelement in der EVENT_QUEUE. Es wird ausgekettet und seine Adresse dem Prozeß übergeben.

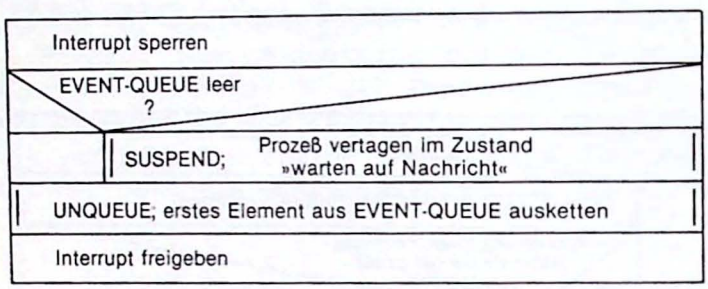

Abb. 3-10: Betriebssystem-Prozedur 'RECEIVE-EVENT'

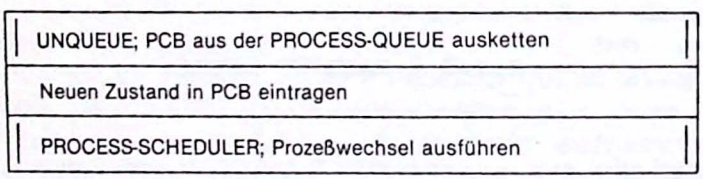

Abb. 3-11: Betriebssystem-Prozedur 'SUSPEND'

3.1.3.3 WAIT_INTERRUPT, SIGNAL_INTERRUPT, DELAY, CONTINUE

Mit diesen Prozeduren werden Prozesse vertagt bzw. fortgesetzt. Beim Vertagen wird der Prozeß in einen bestimmten Zustand versetzt. Bei Aufruf der Prozedur 'WAIT_INTERRUPT' ist der neue Zustand 'Warten auf Interrupt' und bei 'DELAY' ist er 'Warten auf Fortsetzung'. Die Vertagung geschieht durch Aufruf der Prozedur 'SUSPEND', der neue Zustand wird als Parameter übergeben.

Bei Aufruf der Prozeduren 'SIGNAL_INTERRUPT' oder 'CONTINUE' wird die Prozedur 'UNSUSPEND' gestartet. Parameter ist der erwartete Vertagungszustand. Stimmt er überein, so wird der Prozeß fortgesetzt.

3.1.4 Prozeß-Synchronisation

3.1.4.1 PROCESS_SCHEDULER

Die Prozedur 'PROCESS_SCHEDULER' steht dem Anwender nicht zur Verfügung. Die Koordination der Prozesse liegt allein beim Betriebssystem. Deshalb wird der Scheduler auch nur von Betriebssystem-Routinen aufgerufen. Seine Aufgabe ist es, den gerade laufenden Prozeß zu vertagen, d.h. seine Stackregister in dessen Process-Control-Block zu kopieren. Anschließend ist (je nach Scheduling-Methode) z.B. der am längsten wartende Prozeß mit der höchsten Priorität zum Ablauf zu bringen. Seine Stackregister werden dazu vom PCB in den Prozessor geladen. Bei der Auswahl des Prozesses muß der Scheduler die Queue-Header (NUMBER_IN_QUEUE) der einzelnen PROCESS_QUEUEs beginnend bei der mit der höchsten Priorität durchsuchen, bis er eine findet, in die ein Prozeß eingekettet ist. Den ersten Prozeß dieser Queue hat er zu starten (Abb. 3-12), d.h. dessen Stackregister zu laden.

Der ausgewählte Prozeß kann auch wieder der gleiche sein, dessen Stackregister er am Anfang der Prozedur abgespeichert hat, sofern er die in der Scheduling-Methode vorgegebenen Bedingungen erfüllt.

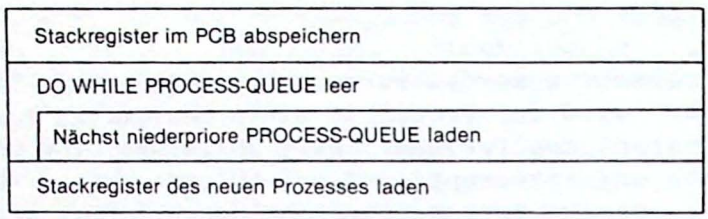

Abb. 3-12: Betriebssystem-Prozedur 'PROCESS-SCHEDULER'

3.1.4.2 SET_PRIORITY, RESET_PRIORITY

Darf ein Prozeß in bestimmten Programmpassagen nicht unterbrochen werden, so hat er die Möglichkeit, den Interrupt zu sperren (DISABLE). Sofern er keine Betriebssystem-Prozeduren aufruft, kann er nicht verdrängt werden. Der Nachteil ist, daß die Interrupts länger anstehen, bis sie angenommen werden. Das Realzeitverhalten kann in bestimmten Fällen gefährdet sein. Außerdem kann es bei zeitkritischen Interrupts, die schnell bearbeitet werden müssen, zu Problemen kommen.

Einige Betriebssysteme bieten dem Anwender deshalb die Routine 'SET_PRIORITY' an (Abb. 3-13). Durch ihren Aufruf wird dem Prozeß momentan eine andere (höhere) Priorität verliehen. Die gewünschte Priorität gibt der Anwender in einem Parameter an. Das Betriebssystem kettet daraufhin den Prozeß in die PROCESS_QUEUE der neuen Priorität ein. Ein Prozeßwechsel kann nicht mehr stattfinden (sofern die neue Priorität hoch genug ist!).

Nach den kritischen Programmphasen, hat der Anwender die Prozedur 'RESET_PRIORITY' aufzurufen (Abb. 3-14). Sie stellt die alten Verhältnisse wieder her, d.h. der Prozeß wird wieder an erster Stelle seiner ursprünglichen PROCESS_QUEUE eingekettet.

In beiden Prozeduren muß der Interrupt während dem Arbeiten mit den Queues gesperrt werden.

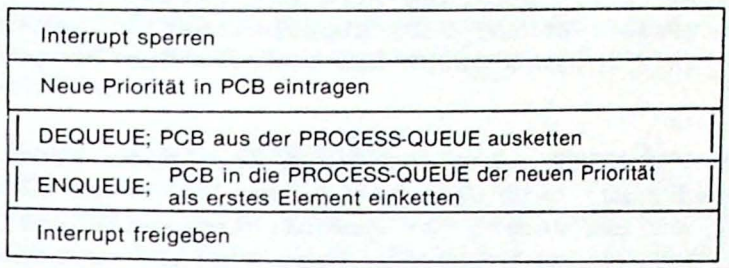

Abb. 3-13: Betriebssystem-Prozedur 'SET-PRIORITY'

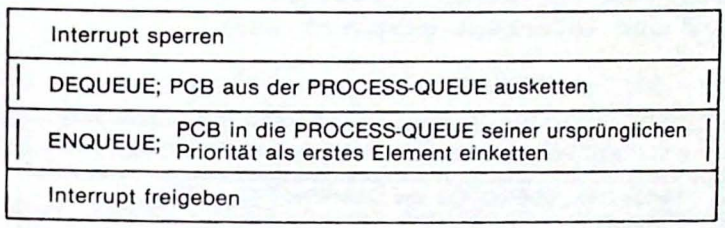

Abb. 3-14: Betriebssystem-Prozedur 'RESET-PRIORITY'

3.1.5 Zeitglieder zuweisen und entlassen

Ein Timer wird von Betriebssystemen, die den Prozessen die Timer nicht fest zuordnen, in der Weise realisiert, daß ein Poolelement gemäß Abb. 2.31 strukturiert und beschrieben wird. Es wird sowohl in die jedem Prozeß zugeordnete TIMER_QUEUE, als auch in die TIME_SLOT_TABLE eingekettet.

3.1.5.1 TIMER_ASSIGN

Möchte der Anwender einen Zeitauftrag abgeben, so ruft er die Betriebssystem-Prozedur 'TIMER_ASSIGN' auf. Als Parameter übergibt er die Anzahl der Zeiteinheiten (TIME), sowie die von ihm definierten Werte für 'TIMER_TYPE' und 'USER_CODE'. Das Betriebs-

system holt sich daraufhin ein Poolelement und trägt 'TIMER_TYPE' und 'USER_CODE' ein. Außerdem wird die Prozeß-Nummer des aufrufenden Prozesses (CURRENT_PID) abgespeichert. Danach wird das Poolelement in die TIMER_QUEUE des aufrufenden Prozesses eingekettet.

In 'TIME_SLOT_POINTER' ist der Index des Time-Slots abgespeichert, dessen Timer beim nächsten Timer-Interrupt ablaufen. Daraus, sowie aus der Anzahl der gewünschten Zeiteinheiten und den maximal zur Verfügung stehenden Time-Slots, errechnet das Betriebssystem den Time-Slot, in den das Poolelement in der Time-Slot-Table eingekettet wird. Der Time-Slot wird ebenfalls in dem Poolelement eingetragen. Es wird bei beiden Verkettungen sowohl vorwärts, als auch rückwärts verkettet (Abb. 3-15). Während des Einkettens muß der Interrupt gesperrt sein.

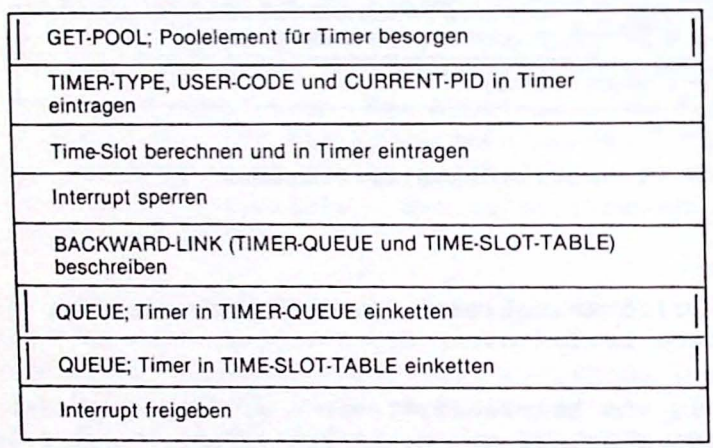

Abb. 3-15: Betriebssystem-Prozedur 'TIMER-ASSIGN'

3.1.5.2 TIMER_RELEASE

Möchte ein Anwender einen von ihm zugewiesenen Timer vor dessen Ablauf löschen, so ruft er die Prozedur 'TIMER_RELEASE' auf. Als Parameter übergibt er den 'TIMER_TYPE' und den 'USER_CODE' des zu löschenden Timers. Das Betriebssystem durchsucht daraufhin die TIMER_QUEUE des Prozesses nach diesem Timer, indem es

'TIMER_TYPE' und 'USER_CODE' der eingeketteten Timer mit den Parametern vergleicht. Hat es den Timer identifiziert, so wird das Poolelement aus der TIMER_QUEUE und dem entsprechenden Time-Slot der TIME_SLOT_TABLE ausgekettet. Es wird anschließend an den Pool zurückgegeben.

Wurde der Timer in der TIMER_QUEUE nicht gefunden, so besteht noch die Möglichkeit, daß sich die Nachricht vom Ablauf des Timers in der EVENT_QUEUE des Prozesses befindet. Eingekettet wird sie dort von dem REAL_TIME_PROCESS wenn der Timer abgelaufen ist. Das Betriebssytem muß also die EVENT_QUEUE nach Nachrichten über abgelaufene Timer durchsuchen. Stimmen 'TIMER_TYPE' und 'USER_CODE' in der Meldung mit den Parametern überein, so muß die Nachricht ausgekettet werden. Das Poolelement wird an den Pool zurückgegeben.

Da der Timer bzw. die Meldung über dessen Ablauf an beliebiger Stelle in den Queues stehen kann, kommt beim Ausketten die Prozedur 'DEQUEUE' zur Anwendung.

Wurde in der EVENT_QUEUE keine Nachricht über den Ablauf des Timers gefunden, so bedeutet das, daß der Anwender einen Zeitauftrag löschen wollte, der nicht (mehr) existiert. In vielen Betriebssystemen führt der Löschauftrag für einen nicht zugewiesenen Timer zum Ablauf einer Fehlerbehandlungsroutine, die alles weitere veranlaßt (z.B. Fehlermeldung über LEDs oder Ausdruck, etc.). Abb. 3-16 zeigt den Ablauf.

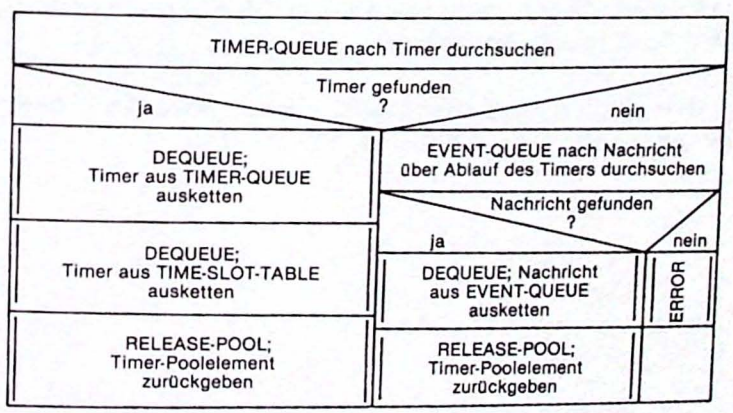

Abb. 3-16: Betriebssystem-Prozedur 'TIMER-RELEASE'

3.1.5.3 Teilqualifizierte Zeitglieder

Es steht dem Anwender vollkommen frei, wie 'TIMER_TYPE' und 'USER_CODE' beschrieben werden. Wie in Kapitel 2.7.1 bereits angedeutet, ist es oftmals sinnvoll, Timer in Gruppen (Typen) zusammenzufassen. Oftmals ist es nötig, alle Timer eines Types zu löschen.

Mit den bisher bekannten Mitteln müßte sich der Anwender merken, welche Timer er zugewiesen hat, die noch nicht abgelaufen sind. Für jeden einzelnen dieser Timer müßte er dann die Prozedur 'TIMER_RELEASE' aufrufen. Sowohl die eigene Timerverwaltung, als auch der u.U. mehrmalige Aufruf von 'TIMER_ RELEASE' kosten Zeit. Es ist deshalb wünschenswert, daß das Betriebssystem den Anwender dahingehend unterstützt.

Das hier beschriebene Verfahren ist als Erweiterungsvorschlag für die meisten Realzeit-Betriebssysteme , die dieses Leistungsmerkmal nicht haben, zu verstehen.

Ziel der Prozedur ist es, daß der Anwender nur einmal 'TIMER_RELEASE' aufzurufen braucht und das Betriebssystem löscht daraufhin alle zugewiesenen Timer des als Parameter angegebenen Types. Der Aufrufer kennzeichnet, daß er diesen Dienst in Anspruch nehmen möchte, indem er als 'USER_CODE' einen dafür vorgesehenen Code (z.B. H'FFFFH) angibt. Das Betriebssystem hat daraufhin die gesamte TIMER_QUEUE und EVENT_QUEUE nach Timern bzw. Nachrichten über deren Ablauf, mit dem angegebenen 'TIMER_TYPE' zu durchsuchen. Bei Übereinstimmung ist das Poolelement auszuketten und an den Pool zurückzugeben.

Danach ist die Suche fortzusetzen, bis jeweils das Ende der beiden Queues erreicht ist (Abb. 3-17).

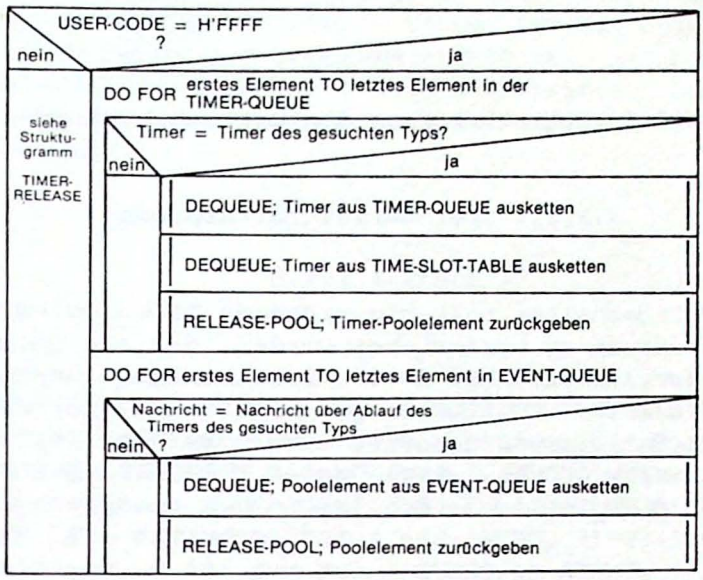

Abb. 3-17: 'TIMER-RELEASE' mit Möglichkeit, alle Timer eines Typs zu löschen

3.2 Systemstart

Bevor nach dem Einschalten die Prozesse gestartet werden können, muß zuerst das Betriebssystem geladen (Code) und initialisiert (Daten) werden. Befindet sich der Code des Betriebssystems auf (P)ROM, so erübrigt sich das Laden. Nach dem Bootstrap kann sofort mit der Initialisierung begonnen werden (Kapitel 3.2.2).

3.2.1 Laden des Betriebssystems

In größeren Systemen befindet sich der Code von Betriebssystem und Prozessen auf einem größeren Speichermedium (z.B. Floppy-Disk). Auf der Rechnerbaugruppe befindet sich nur ein Boot-(P)ROM, das das Laden des Codes von dem Speichermedium in den

RAM-Speicher veranlaßt. Danach erfolgt die Initialisierung des Betriebssystems.

3.2.2 Initialisierung des Betriebssystems

3.2.2.1 Speicherinitialisierung

Der nach dem Einschalten beliebig stehende Datenspeicher muß bei der Initialisierung so beschrieben werden, daß die Queues, Pools, PCBs, etc. entstehen (Abb. 3-18). Die absoluten Adressen wurden beim Binden (LINKER, LOCATER) zugewiesen. Der Interrupt muß während der Initialisierung gesperrt sein. Bei der Initialisierung der Queues (EVENT_QUEUEs, Pool-Queues, PROCESS_QUEUEs, TIMER_QUEUEs, TIME_SLOT_TABLE, etc.) ist in den entsprechenden Queue-Headern in 'FIRST_IN_QUEUE' und 'LAST_IN_QUEUE' die NULL-Adresse einzutragen, 'NUMBER_IN_QUEUE' ist mit '0' zu beschreiben. Die Queues sind somit leer.

Bei der Initialisierung der Pools sind die einzelnen Poolelemente in die Pool-Queues einzuketten.

3.2.2.2 Interrupt-Tabelle

In die Interrupt-Tabelle ist bei den verwendeten Interrupt-Vektoren die Adresse der Interrupt-Programme einzutragen. Es empfiehlt sich, auf nicht verwendete Vektoren die Adresse einer Fehlerroutine zu schreiben. Man vermeidet dadurch ein 'Verlaufen' des Prozessors.

3.2.2.3 PROCESS_LOAD_TABLE

In den PROCESS_CONTROLL_BLOCK sind alle wichtigen Daten (Stack, Priorität) des Prozesses einzutragen. Anschließend ist der PCB in die seiner Priorität entsprechende PROCESS_QUEUE einzuketten,

sofern der Prozeß als im System vorhanden gekennzeichnet ist. Die Zuteilung von Stackadresse, Stackgröße und Priorität der Prozesse obliegt dem Systemprogrammierer. In der PROCESS_LOAD_TABLE (PLT) legt er diese Daten zur Programmierzeit ab. Außerdem ist dort auch die Anfangsadresse der Prozesse gespeichert. Das Betriebssystem liest bei der Initialisierung die PLT und überträgt die Werte in die PCBs.

3.2.2.4 Stack

Die einzelnen Prozesse werden auch beim ersten Mal durch den PROCESS_SCHEDULER gestartet. Der Scheduler lädt die Stackadresse des zu startenden Prozesses. Beim Verlassen der Prozedur (RET-Befehl) wird die auf dem Stack liegende Rücksprungadresse in die Befehlsadressregister übernommen. Der Prozeß wird an dieser Stelle gestartet bzw. fortgesetzt.

Bei der Initialisierung ist deshalb an die entsprechende Stelle in den Stacks der einzelnen Prozesse die Anfangsadresse der Prozesse aus der PLT zu übertragen.

3.2.3 Start der Prozesse

Nachdem alle Datenstrukturen initialisiert sind, lädt das Betriebssystem die Stackadressen des Idle-Prozesses. Anschließend wird der PROCESS_SCHEDULER aufgerufen, der den am höchsten priorisierten Prozeß startet, der zuerst in die PROCESS_QUEUE eingetragen ist.

Nachfolgendes Struktogramm zeigt nochmals den Ablauf der Betriebssystem-Initialisierung im Zusammenhang (Abb. 3-18).

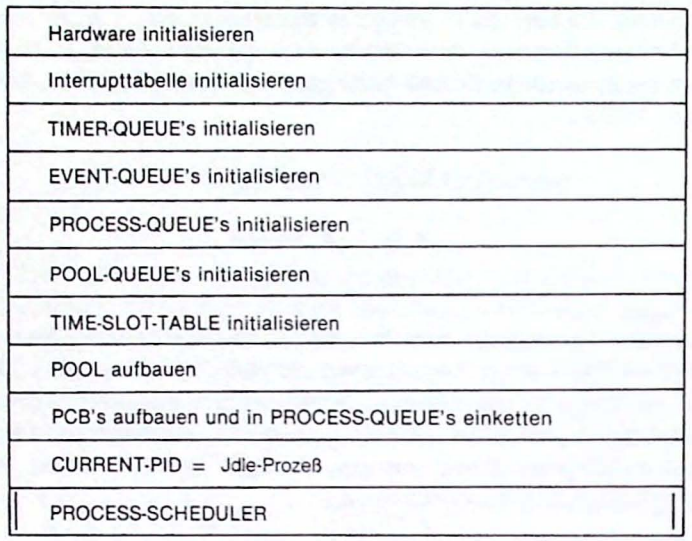

Abb. 3-18: Betriebssystem-Start

3.3 REAL_TIME_PROCESS

Der REAL_TIME_PROCESS gehört zum Betriebssystem und hat die Aufgabe, für die abgelaufenen Timer eine Nachricht an den zugehörigen Prozeß zu senden (Abb. 3-19). Er wird vom Interrupt-Programm des Hardware-Timers angestoßen, d.h. er hat sich mit 'WAIT_INTERRUPT' zu vertagen.

Nachdem er den Zeiger auf den aktuellen Time-Slot (TIME_SLOT_POINTER) inkrementiert hat, muß er alle Timer dieses Time-Slots sowohl aus der TIME_SLOT_TABLE, als auch aus der TIMER_QUEUE des zugehörigen Prozesses ausketten. Er verwendet das Poolelement, das den Timer in den Queues realisiert hat, weiter und schreibt die Nachricht über den Ablauf hinein. Mit SEND_EVENT wird es an den Prozeß versandt. Hat der REAL_TIME_PROCESS die höchste Priorität im System, so kann das Sperren des Interrupts vor dem Ausketten entfallen.

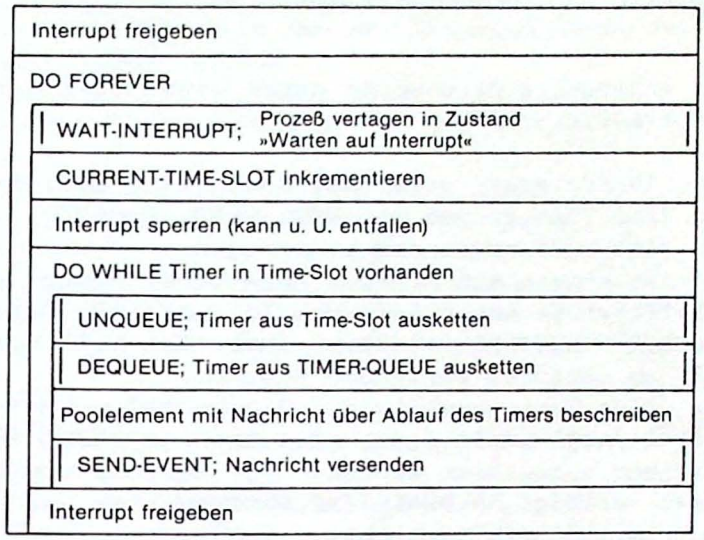

Abb. 3-19: REAL-TIME-PROCESS

3.4 Das Betriebssystem 'ROSY'

Als Beispiel für ein Multitasking-Realtime-Betriebssystem wird in diesem Kapitel das Betriebssystem 'ROSY' (Realtime-Operating-System) und der zugehörige REAL_TIME_PROCESS vorgestellt. ROSY enthält die wichtigsten Prozeduren eines Realzeit-Betriebssystemkerns und ist nach den Struktogrammen der Kapitel 3.1 - 3.2 in PL/M-86 codiert. Der Sourcecode ist im Anhang abgedruckt. ROSY ist ausgetestet und voll funktionsfähig, und es kann vom Leser jederzeit in kleinen Systemen eingesetzt werden. Nach der Lektüre dieses Buches dürfte es auch leichtfallen, zusätzliche Leistungsmerkmale in ROSY einzubringen.

Nachfolgend werden die Leistungsmerkmale und die Benutzerschnittstellen von ROSY beschrieben. Anschließend wird der Weg erläutert, den man gehen muß, um mit ROSY zu einem funktionsfähigen System zu kommen.

3.4.1 Leistungsmerkmale von ROSY

Nachfolgend angeführte Stichworte geben einen Überblick über das Betriebssystem ROSY:

- ROSY kann 10 Prozesse verwalten. Davon sind zwei bereits fest vergeben (für Idle-Prozeß und REAL_TIME_PROCESS), der Anwender kann also 8 Prozesse neu einbringen.
- Die Prozesse können auf 4 Prioritätsebenen gelegt werden. Die 5. Prioritätsstufe ist allein dem Idle-Prozeß vorbehalten.
- Der PROCESS_SCHEDULER startet immer den mit der höchsten Priorität am längsten wartenden Prozeß.
- ROSY hat eine Timerverwaltung, die es jedem Prozeß ermöglicht, 'beliebig' viele Timer zu setzen. Maximal 256 Zeiteinheiten können zugewiesen werden. Die Bearbeitung der gesetzten Timer erfolgt im REAL_TIME_PROCESS. Er wurde auf die höchste Prioritätsstufe gelegt.
- Es stehen 2 Vertagungsmechanismen zur Verfügung, nämlich RECEIVE_EVENT und WAIT_INTERRUPT.
- Die Initialisierung der Prozesse erfolgt über die PROCESS_LOAD_TABLE. Sie muß vom Anwender generiert werden (Kapitel 3.4.3).
- Das Codevclumen von ROSY beträgt ca. 2,7 kByte. Die Größe des benötigten Datenbereichs richtet sich im wesentlichen nach der Anzahl und Größe der Poolelemente und der Größe des Stackbereichs der Prozesse. Sind 128 Poolelemente zu je 27 Byte vorhanden und jeder Prozeß erhält 256 Byte Stack, so hat der Datenbericht eine Größe von ca. 9 kByte.

3.4.2 Benutzerschnittstelle von ROSY

Möchte ein Prozeß die Dienste von ROSY in Anspruch nehmen, so erfolgt dies durch den Aufruf der entsprechenden Betriebssystem-Prozedur. Nachfolgende Liste gibt Auskunft über die implementierten Betriebssystem-Routinen, die dem Anwender zur Verfügung stehen. Die Funktion der einzelnen Prozeduren wird hier nur noch stichwortartig angeführt. Mit welchen Parametern die Prozeduren versorgt werden müssen, kann der Leser dem Sourcecode von ROSY im Anhang entnehmen. Dort ist jede Prozedur mit einem sog. Prozedurkopf (einige Kommentarzeilen) versehen, indem die benötigten

Parameter aufgeführt sind. Das Datenformat der Parameter (BYTE, WORD oder POINTER) kann aus der Datenerklärung am Anfang der Prozeduren entnommen werden.

Folgende Prozeduren von ROSY stehen zur Verfügung:

- QUEUE: Einketten an das Ende einer Queue
- UNQUEUE: Ausketten vom Anfang einer Queue
- ENQUEUE: Einketten in die Mitte einer Queue
- DEQUEUE: Ausketten aus der Mitte einer Queue
- GET_POOL: Anfordern eines Poolelements
- GET_POOL_DIS: Anfordern eines Poolelements aus Interruptprogrammen
- RELEASE_POOL: Rückgabe eines Poolelements
- RELEASE_POOL_DIS: Rückgabe eines Poolelements aus Interruptprogrammen
- SEND_EVENT: Versenden einer Nachricht
- RECEIVE_EVENT: Empfangen einer Nachricht
- WAIT_INTERRUPT: Vertagen, bis Interrupt eintrifft
- SIGNAL_INTERRUPT: Prozeß, der auf den Interrupt wartet, fortsetzen
- SET_PRIORITY: Priorität des Prozesses verändern
- RESET_PRIORITY: Priorität des Prozesses auf seinen ursprünglichen Wert zurücksetzen
- TIMER_ASSIGN: Zeitauftrag abgeben
- TIMER_RELEASE: Zeitauftrag löschen

3.4.3 Systemgenerierung mit ROSY

ROSY und der zugehörige REAL_TIME_PROCESS sind in PL/M-86 geschrieben. Sie müssen auf einem Intel MDS (Microcomputer Development System, identisch mit SME = SIEMENS-Mikrocomputer-Entwicklungssystem) der Serie-III mit dem Compiler PLM86.86 (Version 2.0) compiliert werden.

Um zu einem funktionsfähigen System zu kommen, muß der Anwender als erstes seine Prozesse codieren (in PL/M86 oder Assembler 86) und fehlerfrei am SME übersetzen. Ein Prozeß muß als Prozedur realisiert sein (Beispiel: siehe REAL_TIME_PROCESS). Anschließend muß er alle Moduln (ROSY, REAL_TIME_PROCESS und seine eigenen

Prozesse) binden (LINK86.86) und die absoluten Adressen zuweisen (LOC86.86).

Aus dem Locater-Protokoll kann er die Anfangsadressen seiner Prozesse (d.h. der entsprechenden Prozeduren), sowie des REAL_TIME_PROCESS entnehmen. Er muß jetzt die PROCESS_LOAD_TABLE (PLT) von ROSY beschreiben. An den Prozessen, den Bindeanweisungen und der Zuweisung der absoluten Adressen für den Locater darf jetzt nichts mehr geändert werden.

Die PLT enthält für jeden Prozeß eine Zeile, in die folgende Werte einzutragen sind:

- Der Zustand des Prozesses im System bei der Systeminitialisierung, d.h. ob der Prozeß vorhanden ist oder nicht (PRESENT oder NOT_PRESENT). Für nicht vorhandene Prozesse braucht die PLT nicht weiter ausgefüllt zu werden.
- Die Priorität des Prozesses: Der Idle-Prozeß muß die niedrigste Priorität ('0') erhalten und der REAL_TIME_PROCESS muß auf der höchsten Priorität ('4') liegen. Die übrigen Prozesse können auf die Prioritätsstufen 1-4 verteilt werden.
- Das Codesegment der Anfangsadresse des Prozesses (für Idle-Prozeß nicht nötig).
- Der Offset der Anfangsadresse des Prozesses (für Idle-Prozeß nicht nötig).
- Das Stacksegment des Prozesses.
- Der Stackpointer des Prozesses (Anfangswert).

Für die Zuteilung der Stackbereiche ist im ROSY ein sog. STACK_ARRAY erklärt. Für jeden Prozeß wurden 256 Byte Stack angenommen, STACK_ARRAY hat also eine Größe von 2560 Byte. Der Anwender kann STACK_ARRAY an die Prozesse verteilen. Durch Verändern der Größe in der Datenerklärung im ROSY kann er es leicht seinen speziellen Bedürfnissen anpassen.

Nach dem Beschreiben der PLT ist der Betriebssystem-Modul noch einmal zu compilieren. Anschließend werden die einzelnen Moduln, in genau der gleichen Art und Weise wie vorher, gebunden und die absoluten Adressen zugewiesen.

ROSY ist jetzt mit seinen Prozessen startklar und kann auf einer Prozessorbaugruppe zum Ablauf gebracht werden.

4. Kurzbeschreibung verschiedener Betriebssystem-Kerne

In diesem Abschnitt werden wir auf verschiedene Kerne von Realzeit-Multitasking-Betriebssystemen eingehen. Es handelt sich um käufliche Produkte. Zusätzlich wird der Betriebssystem-Kern "ROSY" mit einbezogen, der im Anhang aufgelistet ist und dessen Benutzerschnittstelle in Kap. 3.4.2 erklärt wurde.

Wir werden nicht versuchen, die einzelnen Betriebssysteme in ihrer vollen Breite zu erläutern. Dazu sei auf die Benutzerhandbücher der Hersteller verwiesen. Es soll auch nicht versucht werden, alle auf dem Markt verfügbaren Betriebssysteme darzustellen. Dazu ist das Gebiet zu stark in Bewegung. Bei der Auswahl haben wir uns auf Betriebssysteme für 16-Bit Prozessoren beschränkt (Auflistung in alphabetischer Reihenfolge).

Auf vergleichende Angaben zur Laufzeit von Systemaufrufen der einzelnen Betriebssysteme haben wir hauptsächlich deswegen verzichtet, weil die Anwenderhandbücher in den meisten Fällen nichts oder nur sehr wenig darüber aussagen. Für das Betriebssytem ROSY sind die Laufzeiten im Codeteil im Anhang dazugeschrieben. Aus unserer Sicht würden wir aber jedem Anwender unbedingt raten, vor dem Kauf eines Betriebssystems diese Werte vom Hersteller oder Distributor abzufordern.

Dabei ist natürlich wichtig, daß auch Vergleichbares miteinander verglichen wird. Als erstes muß die Hardware-Umgebung genau definiert sein, also etwa die CPU-Taktfrequenz oder die Angabe der Wartezyklen. Bei den Systemaufrufen selbst muß die Software-Umgebung beim Aufruf angegeben sein, also etwa ob beim Aufruf ein Prozeßwechsel stattgefunden hat oder nicht.

4.1 iAPX 86/30 Betriebssystem-Prozessor kompatibel mit iRMX 86

Hersteller: Intel

Unterstützte CPU: 8086/88, 80186/88

Benutzerschnittstelle: PL/M-86, Assembler

Größe des Codes: 16 kByte

Prozeß-Koordination: Zusammenfassung mehrer Prozesse zu einem "JOB" möglich. Hierarchische JOB-Strukturen möglich. JOB erzeugen; Prozeß ("Task") erzeugen, löschen, vertagen, wiederaufnehmen, für gewünschte Zeit zur Ruhe setzen.

Prozeß-Kommunikation: Botschaften, kritische Regionen, Interrupt-Prozesse.

Zahl der möglichen Prozesse: Beliebig, Änderung während des Beriebs möglich.

Zahl der möglichen Prioritäten: 256, Änderung der Priorität während des Betriebs möglich

Speicherverwaltung: 1 Speicher-Pool pro JOB; Speichersegmente als Vielfache von 16 Byte zuweisbar.

Zeitverwaltung: Prozeß für gewünschte Zeit zur Ruhe setzen.

Konfiguration: Über unterstützende Software (iOSP 86) am Entwicklungssystem MDS.

Sonstiges: Unterstützung von "Exceptions"; Überprüfung der Systemaufrufe auf Zulässigkeit und Plausibilität. Das Betriebssystem ist zusammen mit Hardware-Funktionen wie Interrupt-Controller, Zeitgeber, Baudratengenerator auf einem Chip integriert. Aufwärtskompatibel zu iRMX 86, dadurch Zugang zu dessen umfangreicher Betriebssystemschale mit I/O-System, Debug-System usw.

4.2 RMOS
MTOS86

Hersteller: Siemens (RMOS), ipi (verfügbar durch Scientific Control Systems GmbH, Hamburg) (MTOS)

Unterstützte CPU: 8086/88

Benutzerschnittstelle: PL/M-86, PASCAL, Assembler

Größe des Codes: 8 kByte

Prozeß-Koordination: Prozesse aktivieren, starten, beenden, beenden mit zeitabhängigem Neustart; Programme mit "überwachtem Zugang" (d.h. Benutzung von nicht reentrantfähigen Programmen durch mehrere Prozesse).

Prozeß-Kommunikation: Botschaften, Semaphore, Ereignis-Flags,

Zahl der möglichen Prozesse: 4096

Zahl der möglichen Prioritäten: 256, Änderung der Priorität während des Betriebs möglich.

Speicherverwaltung: Bis zu 32 Speicher-Pools mit Länge der Poolstücke, die bei Initialisierung des Systems festzulegen ist.

Zeitverwaltung: Zeitintervall abwarten, abbrechen, Ereignisflag nach Zeitintervall setzen, Kalenderfunktion.

Konfiguration: Über unterstützende Software am Entwicklungssystem

Sonstiges: In der Betriebssystemschale sind I/O System, Debugger, Netzwerk-Unterstützung, Multiprozessor-Unterstützung, Gleitkomma-Unterstützung verfügbar. Innerhalb der MTOS Familie sind Betriebssysteme mit ähnlichen Leistungsmerkmalen für eine Reihe weiterer Prozessoren verfügbar.

4.3 RMS 68K

Hersteller: Motorola

Unterstützte CPU: 68000

Benutzerschnittstelle: Assembler

Größe des Codes: 12 kByte

Prozeß-Koordination: Zusammenfassung mehrer Prozesse zu einer "Session". Prozeß erzeugen, löschen, starten, stoppen, wiederaufnehmen, vertagen, für gewünschte Zeit zur Ruhe setzen, Prozeß periodisch aktivieren, logische Verknüpfung zwischen Prozessen herstellen.

Prozeß-Kommunikation: Botschaften, Semaphore

Zahl der möglichen Prozesse: Beliebig, Änderung während des Betriebs möglich.

Zahl der möglichen Prioritäten: 256, Ändernung der Priorität während des Betriebs möglich.

Speicherverwaltung: Speicherstücke aus dem Freispeicher in Vielfachen von 256 Byte zuweisbar. Beschreibung des "Segments" durch Attribute.

Zeitverwaltung: Prozeß für gewünschte Zeit zur Ruhe setzen, Prozeß periodisch aktivieren, Kalenderfunktion.

Konfiguration: Über unterstützende Software am Entwicklungssystem (z.B. EXORmacs).

Sonstiges: Unterstützung von "Exceptions" durch "exception monitor task", Überprüfung der Systemaufrufe auf Zulässigkeit und Plausibilität. Möglichkeit der Prozeß- Aufteilung in "system tasks" und "user tasks". Zeitscheibenbetrieb ist möglich. Erweiterung mit I/O System möglich.

4.4 ROSY

Hersteller: Die Buchautoren, Source-Code im Anhang aufgelistet

Unterstützte CPU: 8086/88, 80186/88

Benutzerschnittstelle: PL/M-86

Größe des Codes: 2.7 kByte

Prozeß-Koordination: Prozesse vertagen, wiederaufnehmen.

Prozeß-Kommunikation: Botschaften, Interrupt-Prozesse.

Zahl der möglichen Prozesse: 10

Zahl der möglichen Prioritäten: 5, Änderung der Priorität während des Betriebs möglich.

Speicherverwaltung: 1 Speicher-Pool mit 128 Poolstücken von je 27 Byte Länge.

Zeitverwaltung: Realtime-Prozeß; beliebig viele Timer pro Anwender-Prozeß möglich, die überlappend laufen können. Zuweisen und Löschen der Timer.

Konfiguration: Siehe Hinweise zur Systemgenerierung in Kap. 3.4.3

Sonstiges: QUEUE-Funktionen sind dem Anwender zugänglich durch die entsprechenden Systemaufrufe; dadurch einfache Handhabung von anwenderdefinierten Warteschlangen.

4.5 ZRTS 8000

Hersteller: Zilog

Unterstützte CPU: Z8001/3, Z8002/4

Benutzerschnittstelle: PASCAL, C

Größe des Codes: 6 kByte

Prozeß-Koordination: Zusammenfassung mehrer Prozesse zu einem "JOB" möglich. Prozeß ("Task") erzeugen, löschen, vertagen, wiederaufnehmen, für gewünschte Zeit zur Ruhe setzen, verriegeln (Prozeßwechsel verhindern), entriegeln.

Prozeß-Kommunikation: Botschaften, Semaphore.

Zahl der möglichen Prozesse: Beliebig, Ändernung während des Betriebs möglich

Zahl der möglichen Prioritäten: 32767, Änderung der Priorität während des Betriebs möglich.

Speicherverwaltung: 1 Speicher-Pool pro Prozeß. Die Größe der Segmente wird bei Initialisierung festgelegt und ist in Vielfachen von 256 Bytes zuweisbar. Die Poolstücke zur Prozeßkommunikation werden von einer speziellen mailbox ("pool-exchange") geholt. Die Länge dieser Poolstücke ist pro mailbox generierbar.

Zeitverwaltung: Prozeß für gewünschte Zeit zur Ruhe setzen. Uhrzeitfunktion. Prozeß bei bestimmter Uhrzeit aufwecken.

Konfiguration: Am Entwicklungssystem über Konfigurationssprache ZCL.

Sonstiges: Unterstützung von "Exceptions" durch Überprüfung der Systemaufrufe auf Zulässigkeit und Plausibilität; Trennung von System Stack und Anwender Stack; Zeitscheibenbetrieb möglich. I/O-System, Debugger und Gleitkomma-Emulator verfügbar.

Dritter Teil:

Systementwurf

5. Systementwurf im Hinblick auf die Umgebung eines Realzeit-Betriebssystems - Hardware Aspekte

5.1 Minimalanforderungen eines Realzeit-Betriebssystems an die Hardware

In diesem Abschnitt wird untersucht, welche Minimaleigenschaften eine Rechner-Hardware haben muß, um ein Realzeit-Betriebssystem sinnvoll einsetzen zu können.

Mit anderen Worten: Ab welcher Größe eines Minimalsystems ist es sinnvoll, mit einem Realzeit-Betriebssystem zu arbeiten ? Diese Grenze ist heute stark im Fließen. Der Grund ist die Entwicklung der Speicher, der Peripheriebausteine und, für Minimalsysteme, der Ein-Chip-Computer. Trotzdem soll versucht werden, einige allgemeingültige Grundvoraussetzungen darzulegen, ohne deren Erfüllung der Einsatz eines Realzeit-Betriebssystems nur schwer vorstellbar ist.

5.1.1 RAM-Speicher

Die Bandbreite der Speicherkapazität für Mikrocomputersysteme reicht (für RAM Speicher) von z.B. 64 KByte für Ein-Chip-Computer bis zu mehreren Megabyte für komplexe Systeme. An dieser Stelle ist der RAM Speicher angesprochen, da jedes Betriebssystem Schreib-Lese-Speicher benötigt. Wo liegt die Untergrenze des RAM-Bedarfs, wenn man annimmt, daß der Programmspeicher (Code) und feste Daten im ROM bzw. PROM oder EPROM liegen (Firmware)?

Speicher-Pool

Zur Kommunikation zwischen den einzelnen Prozessen werden Poolstücke benötigt (s. 2.3). Die Mindestzahl und Mindestgröße der Poolstücke hängt ganz von dem jeweiligen Anwendungsfall ab. Ist der Rechner z.B. in einem Rechnerverbund als Datenübertragungsrechner eingesetzt, so ist mit Lastspitzen in der Datenübertragung zu rechnen, denen in der Zahl der Poolstücke Rechnung getragen werden muß. Bei einem kleinen "stand alone"-System, wie dem

Telefoncomputer aus dem Anfangsbeispiel, müssen nicht so viele Reserven bereitgestellt werden.

Im ersten Fall (Datenübertragungsrechner) kann als Erfahrungswert ca. 50 Poolstücke gelten, im zweiten Fall (Telefoncomputer) wird man mit 20 Poolstücken reichlich bedient sein. Nimmt man (wieder als Erfahrungswert) die Poolstück-Größe mit 30 Byte an, ergibt sich für den Datenübertragungsrechner ein benötigter Speicher Pool von

50 * 30 Byte = 1.5 kByte.

Nimmt man für den zweiten Fall an, daß 10 Byte Poolstück-Größe ausreichen, werden nur

20 * 10 Byte = 200 Byte

für den Speicher-Pool benötigt.

Timer-Tabelle

Werden die Zeitgeberfunktionen, die das Betriebssystem zur Verfügung stellt (s. 2.7), über eine "Time-Slot-Table" abgewickelt, so muß für diese Tabelle Speicher bereitgehalten werden. Beim Aufbau der Time-Slot-Table aus Queue-Headern, z.B. für ein 8086-System mit 10 Byte pro Warteschlangenkopf (First, Last, Number in Queue) und einer maximalen "Timer"-Zeit von 20 Sekunden die z.B. in Schritten von 50 Millisekunden zuweisbar sein soll, müssen

10 * 20 / 0.05 = 4 kByte

Speicher "spendiert" werden. Die Zahl der maximal benötigten Zeitglieder sollte beim Entwurf des Systems abgeschätzt werden, um die dafür erforderlichen Poolstücke von vorneherein in den Speicher-Pool mit einzubeziehen.

Ist der Speicherplatz für die Time-Slot-Table zu groß, hat man z.B. folgende Möglichkeiten zur Reduzierung:

1. Man gesteht jedem Prozeß nur eine eng begrenzte (z.B. 5) Anzahl von Zeitgliedern zu (s. 2.7.2.1). Bei zwei Byte pro Zeitzähler, mit denen dann im Raster von 20 Millisekunden etwa eine Zeit von 20 Minuten zu erreichen wäre, bräuchte man insgesamt nur 10 Byte Speicher pro Prozeß aufzuwenden.

2. Man verfährt wie in Kapitel 2.7.2.4 beschrieben und weist die Zeiten als 16-Bit-Wort zu, von dem das lower Byte gestattet, 256 Time-Slots zu adressieren, und das higher Byte die Anzahl der Durchläufe angibt, bis das Poolelement aus der Time-Slot-Table ausgekettet wird. Dafür ist allerdings im Betriebssystem höherer Aufwand erforderlich. Für ca. 20 Minuten Timerzeit sind dann nur 256 Time-Slots entsprechend 2.5 kByte nötig.

Stack

Ein weiterer "Großverbraucher" von RAM-Speicher Verbrauches ist der Stack, den Prozeduren, also Unterprogramme, verbrauchen. Auf dem Stack werden Rücksprungadressen abgelegt, sowie Prozedurparameter, falls die Prozeduren reentrantfähig sein sollen. Es ist beispielsweise denkbar, daß eine Prozedur unterbrochen wird, durch die Unterbrechung ein Prozeßwechsel stattfindet und der andere Prozeß dieselbe Prozedur benutzt, aber mit anderen Parametern. Dann dürfen die Parameter des alten Prozesses natürlich nicht überschrieben werden, was gewährleistet ist, wenn die Parameter auf dem Stack abgelegt sind.

Der größte Anteil an Stack wird benötigt, wenn Unterbrechungen auflaufen, die sich dann auch noch schachteln können. Bei einer Unterbrechung, die ja zu jeder beliebigen Zeit auftreten kann, müssen alle Register sowie der Zustand des Prozessors, gerettet werden. Für den 8086 heißt das etwa, daß dafür 28 Byte Stack gebraucht werden.

Geht man von der realistischen Annahme von drei geschachtelten Interrupts aus, sind das 84 Byte Stack. Hier sollte man auf jeden Fall einen Sicherheitsspielraum dazugeben und nicht unter 100 Byte Stack vorleisten. Dieser Bereich muß jedem Prozeß zur Verfügung gestellt werden, da ein Interrupt immer einen gerade laufenden Prozeß unterbricht und daher dessen Stack beansprucht. Unter der Annahme von 8 Prozessen für ein kleines System sollte man nicht unter

1 kByte

für den Stack-Bereich bereithalten.

Sonstiges

Außer den bis jetzt genannten Anforderungen braucht das Betriebssystem zusätzlichen RAM Speicher, z.B. für die Prozeß Kontroll Blöcke (s. 2.6.4.1). Bei 22 Byte pro Prozeß ergeben sich unter der Annahme von 8 Prozessen ca.

180 Byte.

Außerdem werden noch Speicherplätze für Warteschlangen-Köpfe ("Queue-Header", s. 2.3) benötigt. Rechnet man 10 Byte pro Kopf, ergeben sich für

8 Prozeß-Eingangswarteschlangen:	80 Byte
8 sonstige Warteschlangen:	80 Byte
	160 Byte.

Addiert man alles, erhält man:

Pool-Speicher:	200 Byte
Timer-Tabelle:	4000 Byte
Stack:	1000 Byte
PCB:	180 Byte
Sonstiges:	160 Byte
Gesamt:	5540 Byte.

Diese Zahl sieht auf den ersten Blick recht bescheiden aus, man muß jedoch bedenken, daß das der Minimaldatenspeicher (RAM) für das Betriebssystem ist. Jeder Anwenderprozeß braucht selbstverständlich ebenfalls RAM Speicher für variable Daten wie Tabellen. Daher wird geraten, den RAM-Speicher, auch für Minimalsysteme, nicht unter 8 kByte auszulegen. Platzprobleme auf der Baugruppe treten dadurch nicht auf, da diese Kapazität heute nur noch einen einzigen Chip benötigt. Die "Preisfrage" wird sicherlich zunehmend in den Hintergrund treten.

Am Speicherplatz zu sparen lohnt sich nicht, denn bei knapp werdendem Speicher ist die aufzuwendende Zeit für die Optimierung der Programme immer ein Vielfaches des Wertes, um den die Hardware teurer gewesen wäre.

5.1.2 ROM-Speicher

Wir gehen davon aus, daß für kleine Systeme der Speicher für Programme und unveränderliche Daten in ROMs abgelegt ist, wobei während der Entwicklungsphase und bei kleineren Produktionsstückzahlen programmierbare und löschbare EPROMs genommen werden. Für einen kleinen Betriebssystem-Kern können ca. drei kByte angesetzt werden. Für acht Anwenderprozesse von je zwei kByte wären dann 19 kByte notwendig. Auch hier wird man sich auf die sichere Seite begeben und mindestens 32 kByte oder mehr vorsehen. Für 32 kByte EPROM-Speicher ist heute ebenfalls nur noch ein Chip nötig.

5.1.3 Zeitgeber

Ein Hardware-Zeitgeber ist eine unabdingbare Voraussetzung für ein Realzeitsystem. Unter einem Hardware-Zeitgeber ("Hardware-Timer") versteht man eine Schaltung, die periodisch einen Impuls abgibt. Die übliche Vorgehensweise ist die, daß bei Eintreffen dieses Impulses ein Interruptprogramm angestoßen wird, welches die nötigen Aktionen vornimmt. Die Periodendauer richtet sich stark nach den Systemanforderungen. Das Zeitraster muß einerseits so fein sein, daß alle anfallenden Realzeitaufgaben erfüllt werden können, andererseits soll es aber nicht feiner als nötig sein, um durch die Zeitgeber-Interrupts das System nicht zu stark zu belasten. Der Zeittakt wird abgeleitet von einer Quarzzeitbasis (z.B 2 MHz), aus der mit der Zeitgeberschaltung die gewünschte Zeit erzeugt wird.

Als Schaltungen werden fast immer hochintegrierte Schaltkreise verwendet, die an den Mikroprozessor-Bus angeschlossen werden können und deren Betriebsarten und Zeiten programmgesteuert in weiten Grenzen variiert werden kann. Neuere CPU's (z.B. Intel 80186) und Ein-Chip-Computer haben oft auch die Zeitgeberschaltung schon integriert. Eine typische Betriebsart, um eine Zeitbasis für ein Realzeitsystem zu gewinnen, ist die Verwendung als periodischer Abwärtszähler. Dabei wird der Baustein mit einem festen Wert vorgeladen, der dann mit der dem Zähler angebotenen Frequenz auf Null heruntergezählt wird. An einem Ausgang des Bausteins wird ein Impuls abgegeben, der als Interruptsignal

verwendet werden kann. Dieser Vorgang wiederholt sich dann beliebig oft.

Wünscht man beispielsweise einen Zeittakt von 4ms und wird der Zählerbaustein bei 2 MHz betrieben, muß der Zähler mit dem Wert 8000 geladen werden, da

$$8000 * 0.5\ s = 4ms.$$

Das Zeitraster richtet sich, wie gesagt, nach den Anforderungen. Sollen Betriebssystem-Zeitaufträge (s. 2.7.1) mit einer Genauigkeit von 20ms zuweisbar sein, müsste unser Zähler mit einem Wert von 40000 vorgeladen werden. 40000 als Hexadezimalzahl ergibt 9C40 oder binär

1001 1100 0100 0000
Bit 15 Bit 0

Es werden also 16 Bit benötigt, um diese Zahl darzustellen. Besitzt der eingesetzte Baustein einen 16 Bit Zähler, kann dieser direkt mit dem gewünschten Wert geladen werden. Hätte der Zähler aber nur 12 Bit zur Verfügung, ist er höchstens auf den Wert 4096 einstellbar.

Zwei Lösungsmöglichkeiten bieten sich an:

1. Hardware

Der Zählerbaustein wird mit einer langsameren Taktfrequenz angesteuert. (Abb. 5-1)

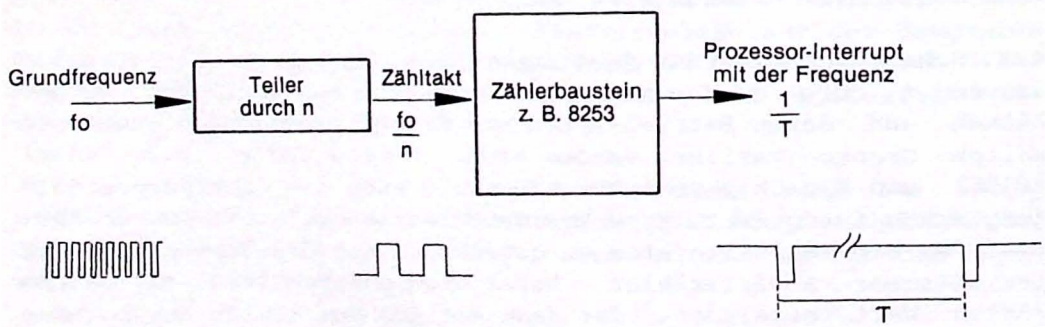

Abb. 5-1: Frequenzteilung durch HW-Vorteiler

Für eine gewünschte Zeitdauer T zwischen zwei Ausgangsimpulsen muß der Zähler mit einem Wert W

$$W = T*f/n$$

geladen werden.

Die Hardware für den Teiler muß sowieso dann vorhanden sein, wenn die Grundfrequenz f der Prozessortakt (z.B 8 MHz) ist, die CPU aber schneller ansteuerbar ist als der Zeitgeberbaustein (z.B 2 MHz).

2. Software

Die andere Lösungsmöglichkeit ist die, daß die Frequenzteilung softwaremäßig vorgenommen wird. Will man eine Zeitbasis von 20ms, könnte man alle 4ms einen Interrupt zulassen, aber nur bei jedem fünften Interrupt das Zeitbearbeitungsprogramm ablaufen lassen.

Das 4ms Unterbrechungsprogramm sähe dann so aus (Abb. 5-2):

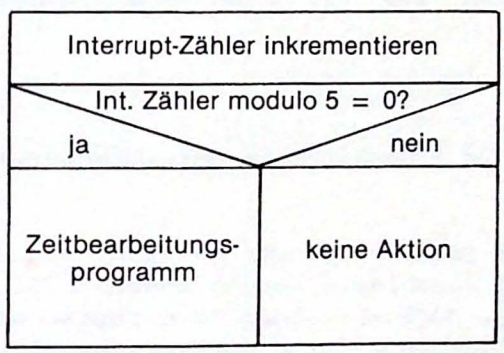

Abb. 5-2: Frequenzteilung durch SW-Teiler

Welche Lösung vorzuziehen ist, muß im Einzelfall entschieden werden. Die "Software"-Lösung hat den Vorteil, daß keine zusätzlichen Hardware-Teiler gebraucht werden. Außerdem können für das Echtzeitverhalten kürzere periodische Zeitintervalle als für die Zeitbasis der Betriebssystem-Zeitgeber nötig sein. Auf Änderungen kritischer Sensoren muß oft schneller als in 20ms reagiert werden. Reichen dafür beispielsweise 4ms , könnte alle 4ms der

Sensor abgefragt werden, und zusätzlich alle 20ms das Betriebssystem-Zeitgeberprogramm im Real-Time-Process (s. 2.8) laufen.

Für 8085/8086 Systeme wird oft der Baustein 8253 als Zeitgeber verwendet. Das ist ein dreifach Zähler-Zeitgeberbaustein mit drei unabhängigen 16 Bit Zählern, die bis zu einer Zählrate von 2 MHz betrieben werden können. Ein anderer oft verwendeter Chip ist der Multifunktionsbaustein 8155, der außer einem 14 Bit Zähler noch RAM Speicher sowie Ein-/Ausgabeports enthält.

5.1.4 Unterbrechungen (Interrupts)

Der Begriff der Unterbrechung ist in den vergangenen Abschnitten oft verwendet worden. Er bedeutet, daß das gerade laufende Programm zu einer beliebigen Zeit unterbrochen werden kann, um dringende, meist zeitkritische Aufgaben auszuführen. Die Möglichkeit zur Unterbrechungsbehandlung wird heute von allen Prozessoren geboten und ist eine unabdingbare Forderung in Realzeitsystemen.

5.1.4.1 Maskierbare Unterbrechungen

Unter maskierbaren Unterbrechungen versteht man solche, die vom Programm gesperrt (maskiert) werden können. Üblicherweise arbeitet die CPU, z.B. 8086 mit einem Interrupt-Controller zusammen, z.B. 8259, der Eingänge für 8 Unterbrechungsquellen hat. Man spricht von Unterbrechungsebenen. Manche CPU's (z.B. Intel 8085) benötigen keinen Interrupt-Controller. An sie können mehrere Unterbrechungsquellen direkt angeschlossen werden. Stehen am Interrupt-Controller mehrere Unterbrechungen gleichzeitig an, muß eine Prioritätslogik entscheiden, welcher Interrupt als nächster angenommen wird. Diese Logik ist z.B. im 8259 enthalten, so daß kein zusätzlicher Hardware-Aufwand nötig ist. Die Vergabe der Interrupt-Prioritäten richtet sich nach Realzeitgesichtspunkten. Interrupt-Quellen, die schnell hintereinander Unterbrechungen produzieren können wie z.B. beim Empfang schneller serieller Daten, brauchen eine hohe Priorität, da der Datenstrom ja nicht abreißt, sondern kontinuierlich von der CPU abgeholt werden muß.

Die Verständigung von Unterbrechungsquelle, Interrupt-Controller und der CPU geht normalerweise nach einem Quittungsverfahren vor sich, wie am Beispiel von 8086 und 8259 erläutert wird:

Steht am Interrupt-Controller eine Unterbrechungsanforderung an, reicht er sie an die CPU weiter, indem er den INTR-Eingang der CPU auf den Pegel "High" legt. Nachdem die CPU mit zwei Pulsen am INTA-Ausgang quittiert hat, übergibt der 8259 ein Byte auf den Datenbus. Der Inhalt wird vom 8086 als Index in eine Tabelle, die sog. Interrupt-Tabelle interpretiert. Aus ihr holt der Prozessor den Interrupt-Vektor, d.h. die Einsprungadresse für das Interrupt-Programm und verzweigt anschließend in dieses Programm (Abb. 5-3).

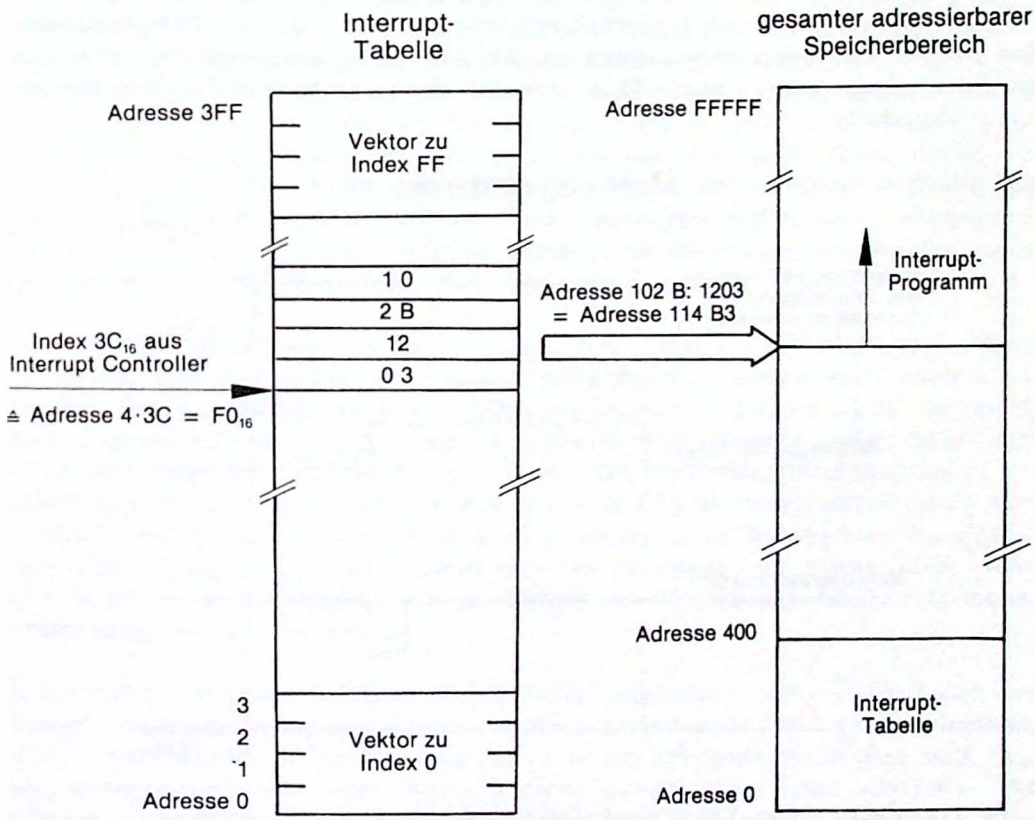

Abb. 5-3: Adressierung von Interruptprogrammen beim Intel 8086

Peripheriebausteine werden häufig als Interrupt-Quellen benutzt. Beispiele sind die serielle Datenübertragung mit den Bausteinen 8251 oder 8274 (Intel) oder das Einschreiben bzw. Auslesen von Daten aus Floppy-Disk-Speichern mit dem Baustein uPD765 (NEC).

Die Peripheriebausteine können häufig verschiedene Funktionen, die voneinander unabhängig sind, ausführen. So kann der Baustein 8274 z.B. auf zwei unabhängigen Kanälen sowohl senden, als auch empfangen und das bei Gleichzeitigkeit aller Funktionen. Gängige Praxis ist, daß nicht für jede Funktion eine eigene Unterbrechungsebene zur Verfügung gestellt wird, sondern für den gesamten Baustein nur ein Interrupt vorgesehen wird. Der Grund des Interrupts, also etwa ob Daten eines Kanales gesendet oder empfangen wurden oder ob ein Fehlerfall vorlag, ist dann durch Auslesen eines Registers im Peripheriebaustein zu erfragen. Das Auslesen des Statusregisters dient dann meist als Quittung von der CPU zum Peripheriebaustein, woraufhin dieser seine Unterbrechunsanforderung wegnimmt.

Ein solcher Vorgang ist in Abb. 5-4 dargestellt.

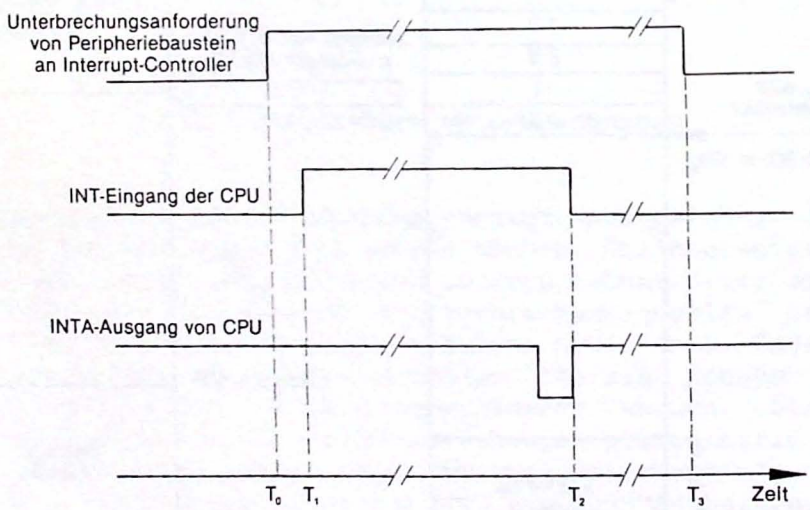

Abb. 5-4: Interrupt-Quittungsverfahren (8086)

Zum Zeitpunkt T0 tritt im Peripherie-Baustein eine Interrupt-Bedingung auf, z.B. der Empfang eines Datenwortes auf einer seriellen Leitung. Zum Zeitpunkt T1 gibt der Interrupt-Controller die Unterbrechungsanforderung an die CPU weiter, die zum Zeitpunkt T2 mit INTA quittiert. Zum Zeitpunkt T3 liest das Interruptprogramm das Statusregister des Peripheriebausteins aus und nimmt dann die Unterbrechungsanforderung zurück.

Die Zeitspanne zwischen T0 und T1 wird normalerweise kurz sein, es sei denn, daß die betreffende Unterbrechungsebene per Programm "maskiert", d.h. abgeschaltet ist, was im 8259 möglich ist. Die Zeit zwischen T1 und T2, also zwischen Eintreffen des Interrupts an der CPU und Quittung von der CPU kann in manchen Fällen sehr lang werden. Läuft beispielsweise die CPU unter Interrupt-Sperre, wie es bei Betriebssystem-Aufrufen nötig ist, wird in dieser Zeit nicht mit INTA quittiert, sondern erst, wenn der Interrupt wirklich von der CPU akzeptiert wurde. Hier zeigt sich die Notwendigkeit, die Zeiten bei Interrupt-Sperren so kurz wie möglich zu halten, um ein günstiges Realzeitverhalten des Systems zu bewahren. Interrupt-Sperren sollten dem Ersteller von Anwenderprogrammen verboten bleiben und nur dem Systemprogrammierer zugänglich sein, um immer einen genauen Überblick über die Programmteile zu behalten, in denen Interrupts gesperrt sind.

Die Zeit zwischen T2 und T3 ist die Zeit, bis das Interrupt-Programm dem Peripheriebaustein quittiert. Diese Zeit kann ebenfalls recht lang werden, nämlich wenn in dieser Zeit weitere, höher priorisierte Interrupts abgearbeitet werden, also wenn sich Unterbrechungen schachteln. Das ist in Realzeitsystemen ein häufiger Fall, der oft schwer zu lokalisierende Fehler nach sich zieht. Hier erhebt sich die Forderung nach äußerster Disziplin bei der Erstellung der Interrupt-Programme; es lohnt sich hier oft noch, in Assembler zu programmieren, um die Laufzeit dieser Programme zu verkürzen.

Eine der angesprochenen Fehlermöglichkeiten ist, daß durch zu lange Unterbrechungssperren Interrupts verloren gehen können, d.h. gar nicht erkannt werden. Die Interrupt-Quelle sei z.B. ein Zeitgeberbaustein, der alle 4 msec einen Interrupt abgibt. Von dieser Zeitbasis sei ein Uhrenprogramm abgeleitet, was die Uhrzeit und das Datum errechnet (s. 2.7.4). Im System seien nun Interrupt-Sperren für diese Unterbrechungsebene vorhanden, die länger als 4 msec dauern, sei es durch fortwährendes Abarbeiten von höher priorisierten Interrupts, sei es daß durch sonstige

lang laufende Programmteile Unterbrechungen generell gesperrt sind. Jetzt passiert es, daß der nächste 4 msec-Interrupt schon ansteht, noch bevor der vorherige abgearbeitet wurde. Geschieht das öfter, führt das dazu, daß der Systemkalender nachgeht.

5.1.4.2 Nicht-maskierbare Unterbrechungen

Obwohl nicht-maskierbare Unterbrechungen nicht unbedingt zu den Minimalanforderungen an eine Hardware zählen, sollen sie hier doch des logischen Zusammenhangs wegen erläutert werden. Eine ausgezeichnete Unterbrechungsebene ist der sog. Non-Maskable-Interrupt (NMI). Er kann nicht gesperrt werden und muß in jedem Fall bearbeitet werden. Er ist vorgesehen, um bei katastrophalen Ereignissen in jedem Fall ein Ausnahme- oder Notprogramm anlaufen zu lassen, unabhängig davon, ob die anderen Unterbrechungen gesperrt sind oder nicht. Beispielsweise kann man sich vorstellen, daß die Spannungsversorgung eine Überwachungseinrichtung besitzt, die einen Impuls abgibt, falls die Versorgungsspannung unter einen vorgegebenen Wert sinkt. Wird dieser Impuls als NMI benutzt, können so bei Spannungseinbrüchen noch wichtige Daten in ein batteriegepuffertes RAM gerettet werden.

Weitere nicht-maskierbare Unterbrechungen sind die sog. "Software-Interrupts". Obwohl sie nicht durch äußere Ereignisse, sondern ausschließlich programmgesteuert aktiviert werden, soll hier kurz auf sie eingegangen werden, da sie vom Prozessor ähnlich wie Hardware-Interrupts behandelt werden (aber z.B. keine INTA Quittung).

Beim 8086 werden Software-Interrupts durch einen eigenen Befehl INT x aktiviert, wobei das x den Index in die Interrupt-Tabelle angibt. Sie werden in Betriebssystemen häufig für den Ansprung der Betriebssystem-Prozeduren, auch Supervisor-Calls (SVC's) genannt, benutzt. Es bestehen gegenüber einem konventionellen CALL folgende Vorteile:

- Die SVC's laufen unterbrechungsgeschützt, wenn nicht Interrupts explizit freigegeben werden.

- SVC's können auch aus Programmteilen, die unter Interrupt-Sperre laufen, aufgerufen werden. Da (z.B. beim 8086) die Prozessor-Flags automatisch beim Einsprung in die SVC-Prozedur abgespeichert werden, bzw. beim Rücksprung (mit IRET-Befehl) zurückgespeichert werden, wird auch automatisch der alte Unterbrechungsstatus beim Verlassen der SVC-Prozedur wiederhergestellt, d.h. ob vorher Unterbrechungen zugelassen bzw. gesperrt waren.

- In neueren Prozessoren laufen Unterbrechungen unter einer anderen Privileg-Ebene als Anwenderprogramme. Da dieses auch für Software-Interrupts gilt, laufen SVC's automatisch ebenfalls privilegiert ab.

Einige Indizes in der Interrupt-Tabelle sind vom Prozessor her vorbelegt und dürfen nicht frei vom Anwender angesprungen werden. Sie nehmen eine Zwischenstellung, zwischen den reinen Hardware- und den reinen Software-Interrupts, ein. Entdeckt der Prozessor z.B. eine Division durch Null, wird ein interner Interrupt mit dem Index 0 beim 8086 erzeugt. Bei dem Vektor den der Prozessor dort findet, sollte dann eine Fehlerbehandlungsroutine abgelegt sein. Man spricht auch von Ausnahmebehandlung oder "Exception-Handling". Bei neueren Prozessoren (z.B.80286), die die Möglichkeit der virtuellen Adressierung bieten, ist eine dieser Exceptions, wenn vom Programm eine Stelle (virtuell) adressiert wird, die im physikalischen Speicher nicht vorhanden ist. Das "Exception-Handling" könnte dann darin bestehen, daß ein SVC anläuft, der die gewünschte Stelle von einem externen Speichermedium, z.B. von einem Plattenspeicher nachlädt. Diese Problematik soll hier jedoch nicht vertieft werden.

5.1.5 Zusammenfassung

Es wird also für Mimimalsysteme folgende Konfiguration vorgeschlagen, ab der ein Realzeit-Betriebssystem sinnvoll eingesetzt werden kann:

- Speicher: 8 kByte RAM,
  32 kByte ROM bzw. EPROM.

- Timer: Ein 16-Bit Zähler, bei Zählfrequenz von mindestens 2 MHz,

- Interrupts: Vier Unterbrechungsebenen, davon je eine für Realzeitinterrupt und Non-maskable-Interrupt.

5.2 Sonstige Hardware-Eigenschaften kleiner Systeme

Die folgenden Hardware-Leistungsmerkmale sind meistens keine unabdingbare Forderung, haben sich aus der Erfahrung jedoch als äusserst nützliche Hilfsmittel für den Betrieb und den Test von Rechnersystemen bewährt. "Für kleine Systeme" soll hier nicht heißen, daß diese Merkmale für große Systeme nicht auch sinnvoll wären. Sie sind jedoch auch und speziell für kleine Systeme, insbesondere im Zusammenspiel mit einem Realzeit-Betriebssystem sehr hilfreich. Bei kleinen Systemen, z.B. Einplatinencomputern wird oft der Fehler gemacht, daß nur das für den späteren Einsatz absolut notwendige an Hardware vorgeleistet wird, ohne beim Entwurf solcher teilweise sehr komplexer Baugruppen die Probleme für die Fehlersuche während der Entwicklungsphase oder die schnelle Funktionsüberwachung im Betrieb zu bedenken.

5.2.1 Anschluß einer Datensichtstation

Datensichtstationen werden oft über serielle, asynchrone Schnittstellen angeschlossen. Der Wert einer solchen Schnittstelle auf kleinen Systemen, z.B. Einplatinencomputern kann nicht genug betont werden. Er liegt darin, daß über die angeschlossene Datensichtstation z.B. Bedienkommandos für den normalen Betrieb möglich sind. Fast noch wertvoller sind solche Schnittstellen jedoch für Diagnose- und Testzwecke z.B. im On-Line Betrieb, wobei natürlich die entsprechende Software-Unterstützung in Form eines On-Line Testsystems vorhanden sein muß.

Verzichtet man auf eine serielle Schnittstelle, oder ist sie aufgrund eines Fehlers nicht mehr bedienbar, kann man sich durch noch einfachere Hardware-Hilfsmittel Zugang zu Betriebs- und Fehlerzuständen verschaffen.

5.2.2 LED-Anzeigen

Für Kontroll- und Wartungszwecke ist es sinnvoll, Anzeigeelemente vorzusehen, die nach außen sichtbar auf einen Blick einen Hinweis auf den Betriebszustand der Baugruppe geben. Wir geben verschiedene Beispiele, wie Anzeigen über den Betriebszustand von Rechnerbaugruppen Auskunft geben können. Selbstverständlich sind hier, je nach Anwendungsfall, beliebige Variationen denkbar.

Fehlerzustände

Bei Verwendung eines Realzeit-Betriebssystems gibt es außergewöhnliche Betreibszustände, bei denen ein sinnvoller Betrieb nicht mehr möglich ist. Drei Beispiele seien angeführt:

Stacküberlauf eines Prozesses

Verbraucht ein Prozess mehr Stack-Bereich als ihm zugewiesen wurde z.B. durch Schachtelung zu vieler Interrupts, wird möglicherweise ein Speicherbereich beschrieben, der für ganz andere Zwecke vorgesehen war.

Poolmangel

Werden z.B. in einer Überlastsituation alle vorhandenen Poolstücke verbraucht, ist ein sinnvoller Betrieb nicht mehr möglich. Derselbe Fall tritt auf, wenn ein Anwender vergißt, die vom Betriebssystem bekommenen Poolstücke wieder zurückzugeben.

Speicherfehler

Es wird oft praktiziert, daß bei ROM-Speichern permanent ein Speichertest im Hintergrund durchgeführt wird. D.h. bei Programmierung des Speichers wird an eine bestimmte Stelle eine Prüfsumme geschrieben. Der Speichertest besteht darin, daß diese Prüfsumme ständig neu errechnet wird und mit der eingeschriebenen verglichen wird. So können etwa "weiche" Speicherzellen, die ihren Inhalt wegen defekter Bausteine verlieren, erkannt werden.

In allen drei Beispielen sind verschiedene Reaktionsmöglichkeiten denkbar. Man kann einen Neustart durchführen, oder bei Hardware-Defekten das Programm in eine Endlosschleife laufen lassen usw. Auf jeden Fall ist es sinnvoll, dem Betreiber einen Hinweis auf den aufgetretenen Fehler zu geben. Man kann z.B. auf der Blende der Rechnerbaugruppe eine 7-Segmentanzeige anbringen, in die beim Erkennen des Fehlers ein Code geschrieben wird, der den Fehler anzeigt. Ebensogut sind natürlich einfache LEDs geeignet, wenn man sich den 7-Segment Dekoder sparen will.

Anzeige für Prozessorbelastung

Ist in einem Rechner zu einer Zeit kein Anwenderprozeß aktiv, läuft ein Prozeß niederster Priorität, der sog. "Idle"-Prozeß, in dem z.B. die erwähnten Speichertests ablaufen können. Ein Maß für die Belastung des Rechners durch Anwenderprozesse ist es, wie oft in einer vorgegebenen Zeit der "Idle"-Prozeß läuft. Diese Tatsache kann man optisch anzeigen, indem man z.B. in jedem hundertsten Durchlauf des "Idle"-Prozesses eine Leuchtdiode abwechselnd ein- und ausschaltet. Bei hoher Prozessorbelastung wird dieser Blinkrhythmus dann verlangsamt und hört eventuell ganz auf. Falls ein Anwenderprozeß die Maschine blockiert, z.B. durch eine fehlerhafte Endlosschleife, würde das ebenfalls durch Beenden des LED-Blinkens angezeigt werden.

Hardware-Zustände

Hier kann man Hardware-Zustände der CPU anzeigen, z.B. ob die CPU im HALT-Zustand steht oder nicht. Hat ein Rechner Geräte zu bedienen, ist eine Anzeige über die Zustände der angeschlossenen Geräte möglich. Handelt es sich z.B. um serielle synchrone Schnittstellen, über die etwa ein HDLC Protokoll abgewickelt wird, können LEDs über die physikalischen oder logischen Zustände der Leitung Auskunft geben, z.B:

LED aus: noch kein Datenverkehr aufgenommen.
LED ein: Datenverkehr zur Gegenstation läuft.
LED blinkt: Datenverkehr lief und ist zusammengebrochen.

Diese drei Punkte sollten lediglich Hinweise geben, wie einfache Leuchtdioden nützlich eingesetzt werden können. Solche Anzeigen kosten kaum Hardware-Aufwand, da LEDs meist direkt von Portbausteinen angesteuert werden können. Leider wird diese Möglichkeit

von Anbietern kleiner Systeme (Einplatinencomputer) meist nicht berücksichtigt.

Oft wünscht man sich nicht nur passive Anzeigen von Hardware- oder Software-Zuständen, sondern auch eine aktive Überwachung im Betrieb. Eine solche, rein hardwaremäßige, Einrichtung ist ein Watchdog.

5.2.3 Watchdog

Ein sog. Watchdog (=Wachhund) ist eine Hardware-Einrichtung, mit der der Ablauf der Software überwacht wird. Grob skizziert, arbeitet ein Watchdog folgendermaßen:

Ein Hardware-Zähler wird in bestimmten Programmteilen immer wieder aufgezogen, so daß er im regulären, fehlerfreien Betrieb nie ablaufen würde. Das Ablaufen des Zählers wird als nichtbehebbarer Fehler gewertet und bewirkt z.B., daß die Baugruppe nur nach dem Ausschalten und Wiedereinschalten der Versorgungsspannung wieder in Betrieb gehen kann.

Was wird mit einem Watchdog kontrolliert?

> Fehlerhafte Endlosschleifen in Prozessen, in denen sich dann der Prozeß nie vertagt. Das führt dazu, daß dieser Prozeß, je nach Scheduling Strategie (s. 2.6.4.2) nur noch von höher-priorisierten Prozessen verdrängt wird und so insbesondere der Idle-Prozeß nie wieder läuft.
>
> Adressierungsfehler in Programmen, die bewirken, daß das Programm an unsinnige Stellen springt und dann nur noch durch großen Zufall wieder an die Stelle kommt, wo der Watchdog Zähler wieder aufgezogen wird.

An welcher Stelle im Programm ist ein Aufziehen des Watchdog-Zählers, der sog. "watchdog retrigger" sinnvoll?

Um mit einem Watchdog alle Prozesse, auch die von niederer Priorität, kontrollieren zu können, ist eine mögliche Stelle des Aufziehens der Idle-Prozeß. Diese Möglichkeit hat den Nachteil, daß in Lastsituationen, in denen ja auch der Idle-Prozeß längere

Zeit nicht an die Reihe kommen kann, der Watchdog ablaufen kann. Eine andere Möglichkeit des Retriggers ist das Betriebssystem, und zwar an der Stelle im Scheduler, wo ein Prozeßwechsel veranlaßt wird. Diese Methode läßt zwar den Watchdog in Überlastsituationen nicht ablaufen, Fehler in niedrig-priorisierten Prozessen werden jedoch nicht erkannt, falls häufig Prozeßwechsel stattfinden.

Da ein Ablaufen des Watchdog immer als katastrophales Ereignis gewertet wird, werden Watchdogs oft mehrstufig ausgeführt. Das soll heißen, daß nach erstmaligem Ablaufen das Programmsystem noch eine "Chance" bekommt und von einem definierten Punkt neu anläuft. Im einfachsten Fall wird der Rechner zurückgesetzt. Spricht der Watchdog danach in einer bestimmten, längeren Zeit nochmals an, sorgt eine Hardware-Schaltung dafür, daß nur noch nach Spannungs-Reset der Betrieb wieder möglich ist (vgl. Abb. 5-5).

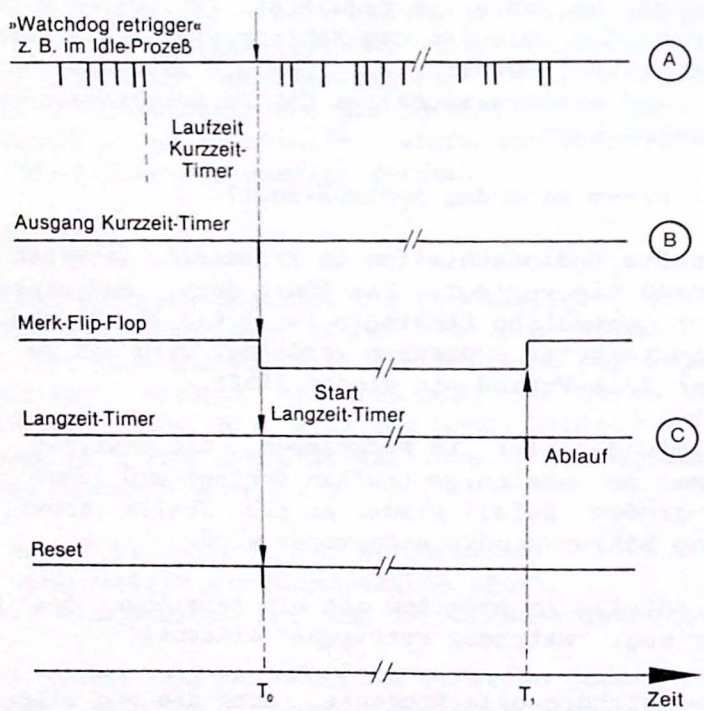

Abb. 5-5: Watchdog: Kurzzeit-Timer läuft einmal ab

Der in Abb. 5-5 skizzierte Watchdog arbeitet folgendermaßen:

Ein Zählerbaustein (z.B. 8253) wird vom Programmteil "watchdog retrigger" ständig neu aufgesetzt (Zeile A). Zur Zeit T0 laufe wegen eines Software-Fehlers der Zähler 1 ab. Das von diesem Zähler erzeugte Ausgangssignal (Zeile B) wird als Startimpuls für einen zweiten Zähler, der auf eine längere Zeit aufgesetzt ist, gewonnen.

Gleichzeitig passieren zwei weitere Aktionen:

1. Dieser Ausgangsimpuls wird als Non-maskable-Interrupt (s. 5.1.4.2.) genommen, der beispielsweise dazu führt, daß der Rechner über Reset wieder hochläuft.

2. Der Ausgangsimpuls triggert ein Merk-Flip-Flop, dessen Ausgangszustand besagt, daß der Kurzzeit-Timer (Zähler 1) abgelaufen war.

Läuft nach Reset das Programmsystem wieder ordnungsgemäß an (Zähler 1 läuft nicht mehr ab), läuft nun dafür der Langzeit-Zähler ab (Zeit T1 in Zeile C). In diesem Fall wird das Merk-Flip-Flop zurückgesetzt und es ist wieder der alte, fehlerfreies Arbeiten anzeigende Watchdog-Zustand eingestellt.

Läuft nach Reset das Programm nicht wieder sauber an (Zähler 1 läuft sofort wieder ab, s. Abb. 5-6), werden die beiden Tatsachen

Ablauf Zähler 1 und
Merk-Flip-Flop gesetzt

dazu verwendet, durch eine geeignete Schaltung die Baugruppe zu verriegeln. Eine Möglichkeit ist, sie im Reset Zustand festzuhalten, aus dem sie dann erst wieder durch Aus- und Einschalten der Versorgungsspannung wieder herauskommt.

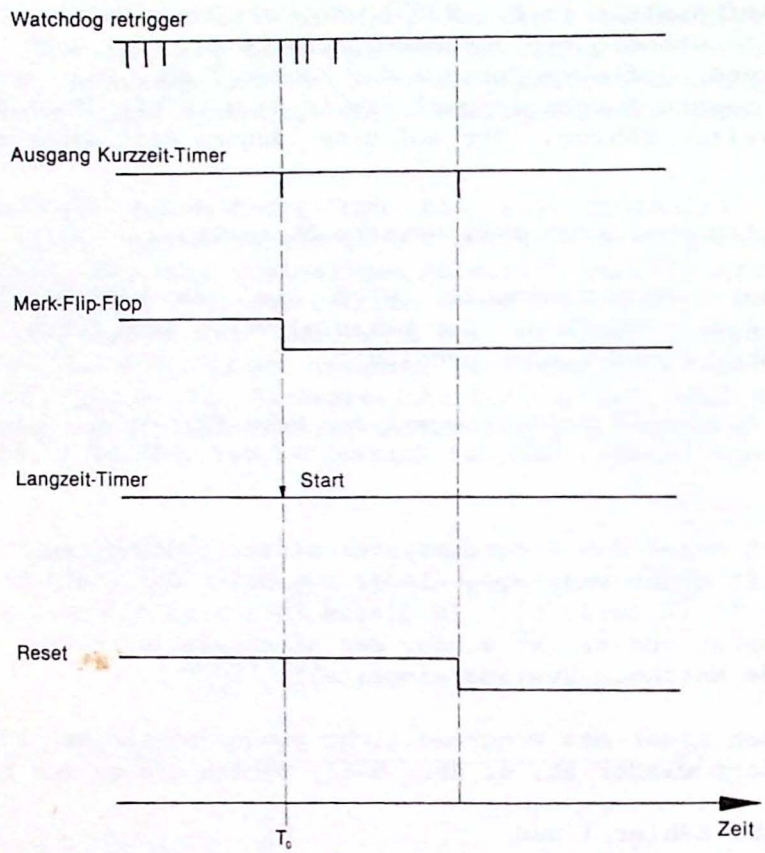

Abb. 5-6: Watchdog: Kurzzeit-Timer läuft ab, während Langzeit-Timer läuft

Damit das so funktioniert, darf der Langzeit-Zähler während seines Laufes nicht neu anstoßbar sein, also insbesondere nicht aufgezogen werden, während das Programm nach Zähler 1 Ablauf über Reset läuft. Dazu muß eine Hardware-Schaltung verfügbar sein, die es gestattet, programmgesteuert den Langzeit-Zähler zu verriegeln.

Der hier beschriebene Watchdog ist natürlich nur eine von vielen Realisierungsmöglichkeiten. Sinnvolle Zeiten sind für den Zähler 1 etwa 100ms und für den Langzeit-Zähler einige Minuten.

Damit ist die Behandlung der Punkte, die uns für kleine Systeme wichtig erschienen, abgeschlossen. Wir haben uns auf die Teile beschränkt, die allgemeineren Charakter haben. Im folgenden Abschnitt wird ein Beispiel eines Einplatinencomputers gezeigt, auf den viele der erwähnten Eigenschaften für kleine Systeme zutreffen.

5.2.4 Realisierungsbeispiel

Die hardwaremäßigen Möglichkeiten zur Realisierung von Rechnersystemen sind heute sehr vielfältig. Es gibt käufliche Baugruppen-Systeme, die für Standardanwendungen eine Hardware-Entwicklung überflüssig machen. Beispiele sind das SMP-System (Siemens), das SBC-System (Intel) oder die ELZET-80 Europakartenserie (Kontron).

Wir wollen diese Systeme hier nicht weiter diskutieren, sondern stattdessen einen Einplatinencomputer zeigen, der nur für einen bestimmten Anwendungsfall im Rahmen eines größeren Systems eingesetzt wird und auf den viele der erwähnten Anforderungen für kleine Systeme zutreffen.

Abb. 5-7 zeigt das Blockschaltbild, Abb. 5-8 die Ausführung.

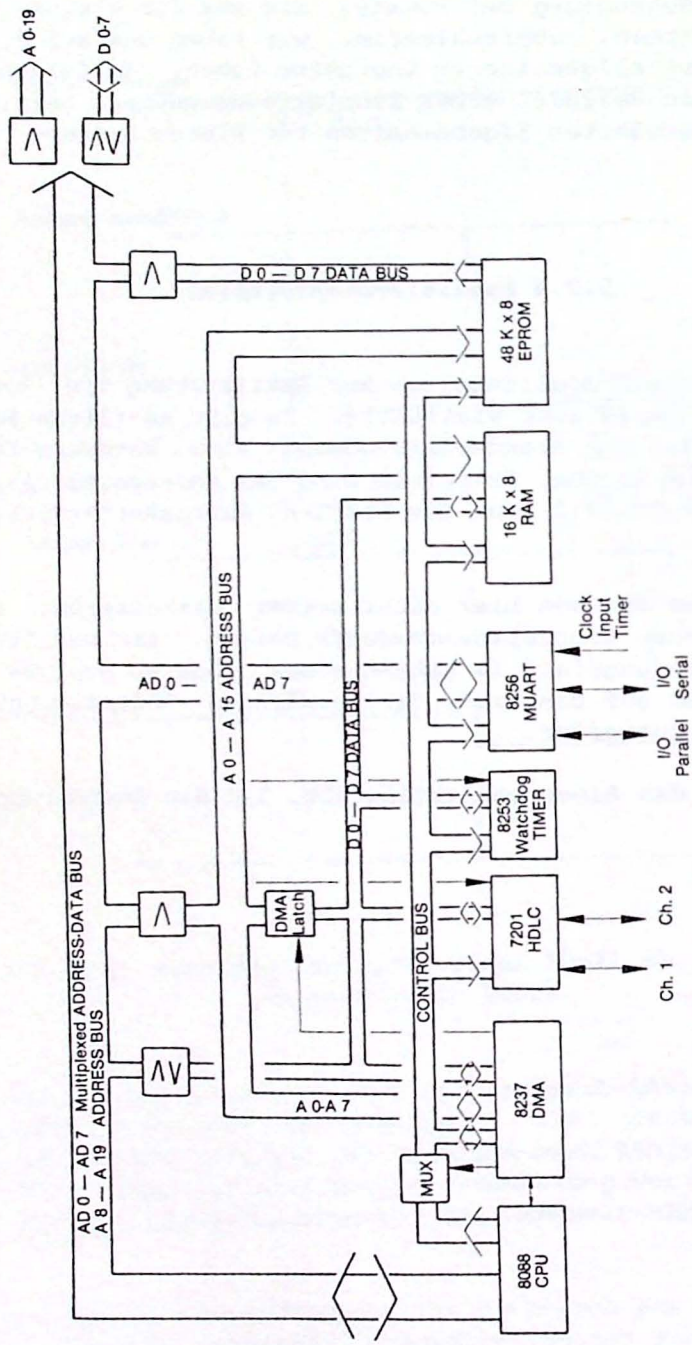

Abb. 5-7: Blockschaltbild Peripheriebaugruppe

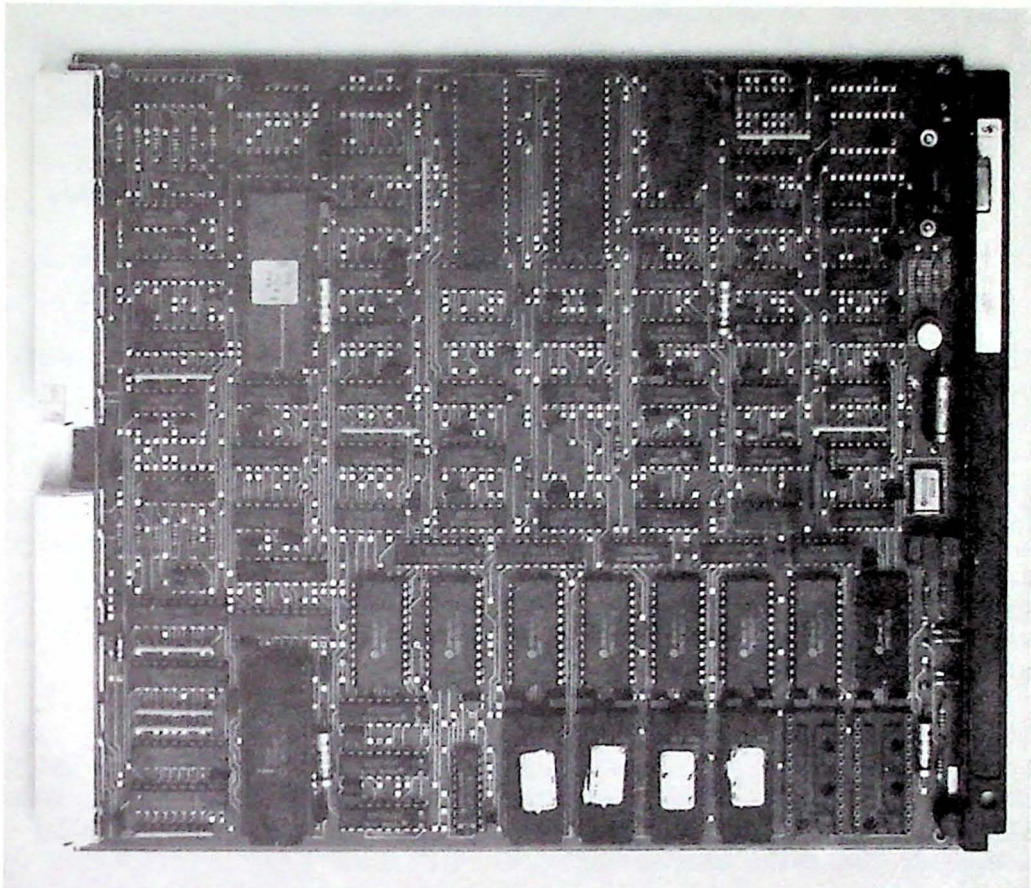

Abb. 5-8

Die Aufgabe des Rechners besteht darin, die von einer separaten Hardware in einen Pufferspeicher abgelegten Zustände von Überwachungssensoren aufzunehmen und nach entsprechender Verarbeitung über die beiden HDLC-Datenstrecken an einen "Host"-Rechner weiterzumelden. Der Pufferspeicher ist als eigene Baugruppe ausgeführt. Die gezeigte Rechnerbaugruppe besteht aus folgenden Teilen:

CPU: 8088, Minimum Mode (Intel)

Speicher:	48 kByte EPROM 2764 16 kByte statisches CMOS-RAM
Peripherie:	MPSC (Multi Protocol Serial Controller) 7201 (NEC) als HDLC Baustein mit 2 unabhängigen Kanälen, Datenrate 800 kBit/sec; DMA-Controller 8237 für direkten Speicherzugriff von 7201. 4 Kanäle, 2 für MPSC senden, 2 für empfangen; MUART 8256 (Siemens):Unterbrechungssteuerung, serielle asynchrone Schnittstelle für Datensichtgerät, Ein-/Ausgabeport zu Steuer- und Kontrollzwecken, Zeitgeber für 4ms-Interrupt. Timer 8253 (Intel): Zeitgeber für die Watchdog-Logik.

An der Frontblende befinden sich der Reset-Schalter, zwei LEDs und der Anschluß für das Datensichtgerät. Die untere LED zeigt den CPU-Zustand an. LED ein: CPU läuft. LED aus: CPU steht im Halt. Die obere LED zeigt den Zustand des Watchdog an: LED ein: Watchdog ist abgelaufen, Rechner kann nur durch Spannungs-Reset wieder gestartet werden.

Die beschriebene Baugruppe hat verschiedene Ein-/Ausgabemöglichkeiten. Die serielle, asynchrone Schnittstelle wurde schon kurz angesprochen. Im folgenden Kapitel soll versucht werden, etwas allgemeiner auf einige weitere Fragen von I/O-Systemen aus Blickwinkel der Hardware einzugehen.

5.3 Ein-/Ausgabe-Systeme

Ein-Ausgabe-Systeme stellen die Verbindung eines Rechners zu externen Geräten dar. Das können z.B. Datensichtstationen, Massenspeicher, Drucker usw. sein, aber etwa auch Sensoren, Anzeigen, LED's usw. Hier soll nicht versucht werden, eine vollständige Aufzählung und Beschreibung von I/O-Systemen zu geben.Vielmehr ist es das Ziel, aus Sicht der Hardware einige exemplarische Eigenschaften zu beschreiben, die zum effektiven Betrieb eines Rechners, insbesondere unter der Regie eines Realzeit-Betriebssystems, wichtig sind. Das hier aus Hardware-Sicht Beschriebene

ist ein Teil des kompletten I/O-Systems der Betriebssystem-Schale (s. 1.1.2), welche die Gesamtheit von Hardware und Software umfaßt, um Anwendern komfortable Schnittstellen zur Peripherie zu bieten.

5.3.1 Eingabe von Sensoren unter Realzeitgesichtspunkten

Realzeitfähigkeit eines Systems bedeutet, in einer vorgegebenen, maximalen Zeit auf äußere Veränderungen reagieren zu können. Diese Veränderungen können sich auf die verschiedensten Werte beziehen. Es können Werte von Temperatur- oder Drucksensoren sein, es kann die Erfassung von kurzen Zeitintervallen gemeint sein (Stoppuhren), es können vom Menschen verursachte Manipulationen sein (z.B. Abheben eines Telefonhörers), es können Wegänderungen sein (z.B. Fortschreiten eines Fließbandes). Der Katalog ließe sich beliebig fortsetzen.

Bei der Festlegung des Systemkonzeptes sollte man einige Fragen beantworten, die die Realzeitfähigkeit des Systems betreffen:

1. Welches ist die maximal geforderte Reaktionszeit?

2. Wieviele der Sensoren müssen diesen absoluten Grenzwert erfüllen, um z.B. Beschädigungen an Anlagen zu vermeiden?

3. Für wieviele der Sensoren sollte im Normalfall diese Maximalzeit erfüllt werden bzw. kann in besonderen Fällen (z.B. Überlast des Rechners) diese Zeit überschritten werden, ohne daß katastrophale Fehlfunktionen eintreten?

4. Reicht die Abtastgeschwindigkeit des Rechners aus, um in der geforderten Zeit die gewünschte Zahl von Sensoren zu erfassen?

5. Reicht die Rechengeschwindigkeit aus, um das System in der geforderten Zeit reagieren zu lassen?

Zwei Beispiele sollen dieses Thema aus Hardware-Sicht beleuchten:

1. Wenige Sensoren mit kurzer Reaktionszeit

Soll ein Leitungspaar auf Kurzschluß zwischen den Adern überwacht werden, kann eine Reaktionszeit von wenigen Millisekundens notwendig sein, um Zerstörungen zu vermeiden. Die Überwachung kann durch Messen des Stromes auf der Leitung, also etwa durch Spannungsabfall über einem bekannten Widerstand erfolgen. Der Analog-Digital-Wandler muß alle paar Millisekunden vom Rechner gelesen werden, was durch eine Unterbrechung geschehen kann. Das entsprechende Interrupt-Programm muß natürlich so ausgelegt sein, daß der Rechner auch in dieser Zeit die Aufgabe bewältigen kann. Auf einige der entstehenden Software-Probleme wird in Kapitel 6.2.4.1 eingegangen werden.

2. Viele Sensoren mit längerer Reaktionszeit

Sind viele Sensoren abzufragen (z.B. 100 und mehr) kann man bei dieser Vielzahl von Abfragestellen nicht mehr jeder eine eigene Unterbrechungsleitung zugestehen. Die Sensoren werden üblicherweise an Ports angeschlossen, die zyklisch abgefragt werden müssen. Manche CPUs bieten die Möglichkeit vom Befehlssatz her, ganze Speicherbereiche sehr schnell zu vergleichen (Beispiel 8086: String-Befehle). In diesem Fall ist es sinnvoll, einen FIFO-Speicher zu verwenden. Am Eingangsport trägt eine Hardware-Logik in zyklischen Abständen die Werte der Sensoren ein, am Ausgangsport werden die Werte vom Programm ausgelesen. Bei Verwendung eines solchen Pufferspeichers sind die erreichbaren Abfragezeiten etwa 5ms für 500 Sensoren. Das ist jedoch die "nackte" Zeit für das Erkennen einer einzigen Änderung der 500 Sensoren.

5.3.2 Serielle Schnittstellen

Serielle Schnittstellen für asynchrone Datenübertragung werden auf Rechnerbaugruppen oft zum Anschluß von Datensichtstationen oder Bedienblattschreibern benutzt. Die Funktionsweise dieser Datenübertragungsmethode wird als bekannt vorausgesetzt. Wird der Rechner über die Sichtstation im Normalbetrieb gefahren, z.B. als Betriebsterminal, von dem aus Bedienkommandos eingegeben werden können, ist eine Standard-V.24-Schnittstelle vorzusehen. Dann kann das Terminal bis zu einer Entfernung von etwa 20m sicher

betrieben werden. Sind größere Entfernungen vorgesehen, muß der Anschluß über ein Modem vorgenommen werden.

Oft reichen jedoch Entfernungen zum Datensichtgerät von ca. 2 Metern. Dann genügt es, auf der Rechnerbaugruppe lediglich eine TTL-Schnittstelle zu realisieren, über die der serielle Datenstrom z.B. direkt von einem USART-Baustein 8251 geleitet wird. So spart man sich die ±12 Volt Spannungsversorgung der V.24 Schnittstelle. Dieses Vorgehen ist besonders dann sinnvoll, wenn das Datenterminal nur gelegentlich zu Diagnosezwecken benötigt wird, z.B. zur Fehlersuche im On-Line Betrieb.

Wenn eine Datensichtstation bedient wird, ist der zeitkritische Teil einer solchen Schnittstelle die Ausgabe, da oft lange Datenfelder durchgehend auszugeben sind. Daher wird man während der Ausgabe nicht den ganzen Rechner blockieren, sondern nach jedem auszugebenden Zeichen vom Schnittstellenbaustein einen Interrupt an die CPU abgeben, um ihr zu signalisieren, daß das nächste Zeichen gesendet werden kann. In welchen Zeitabständen die Ausgabeunterbrechungen kommen, zeigt für verschiedene Datenraten die Tabelle 5-1:

Tabelle 5-1

Datenrate (Bit/sec)	Zeit pro Zeichen (10 Bit) im msec
200	50
600	16.7
1200	8.3
9600	1.0
19200	0.5

Die Eingabe von einer Datensichtstation ist vom Zeitverhalten her unkritisch, da der Benutzer durch die Bedienung einer Tastatur keine hohen Datenraten erzeugen kann. Daher kann die Eingabe im "Polling" Betrieb erfolgen. Das heißt, daß in periodischen Abständen der Schnittstellenbaustein abgefragt wird, ob eine Eingabe erfolgt ist. Diese Methode bietet sich an, wenn keine Unterbrechungsebene am Interrupt Controller mehr frei ist.

Ganz anders sieht die Lage aus, wenn kein menschlicher Benutzer die Eingabedaten erzeugt; sondern ein stetiger Datenfluß vorliegt, z.B. von einem Partner-Rechner. In diesem Fall ist die

Empfangsseite kritischer als die Sendeseite, da die Daten vom Rechner spätestens dann abgeholt sein müssen, wenn der interne Puffer des Peripheriebausteins voll ist. Hier sollte man unterbrechungsgesteuert arbeiten. Erlaubt es die Datenrate, kann man auch im Interrupt-Programm des Zeitinterrupts den Peripheriebaustein pollen. Der Nachteil ist, daß die Grundlast des Rechners steigt, da immer gepollt wird und nicht nur bei Bedarf. Diese Methode ist natürlich auch bei der Ausgabe möglich.

Bei sehr hohen Datenraten ist auch der unterbrechungsgesteuerte Datentransfer noch zu langsam, wie folgende einfache Betrachtung zeigt.

5.3.3 Geräte mit hohen Datenraten

Bei Ein/Ausgabegeräten, die zum Betrieb höhere Datenraten erfordern, z.B. Diskettenspeicher, reicht die interruptgesteuerte Methode des Datentransfers nicht mehr aus. Ein gängiger 8-Zoll Diskettenspeicher einfacher Schreibdichte erfordert eine Datenrate von 250 kBit/sec, was bei Übertragung von 8 Bit-Worten über einen Floppy-Disk-Controller, z.B. uPD765 von NEC ca. alle 30 usec ein geschriebenes bzw. gelesenes Byte bedeutet; bei doppelter Schreibdichte verdoppelt sich auch die Datenrate.

Da jedes Interruptprogramm, das asynchron, d.h. zu beliebigen nichtvorhersagbaren Zeiten angestoßen werden kann, immer alle benötigten Register und den Prozessorstatus speichern muß, bedeutet das, je nach CPU, einen Programmüberbau, der bei 60 usec liegt.

In diesen Fällen muß man das Prinzip des direkten Speicherzugriffes (DMA) anwenden. Aber auch das bedeutet (ganz abgesehen von dem Hardware-Aufwand) noch eine erhebliche Belastung des Rechners, da (z.B. bei einer 5 MHz 8086 CPU) jeder DMA-Zyklus ca. 3 usec benötigt, für die die CPU den Rechner-Bus nicht benutzen kann. Bei 8" Disketten doppelter Schreibdichte bedeutet das während der Datenübertragung eine Busbelastung von 20%. Rechnet man den Software-Aufwand für Dateiverwaltung dazu, sieht man, daß kleine Systeme hier schnell an ihre Grenzen stoßen, wenn sie außerdem gleichzeitig noch andere Aufgaben erledigen sollen. Hier ist der Gebrauch eines eigenen Prozessors, der z.B. nur für die

Ein-/Ausgabe zuständig ist, anzuraten. Man hat ein Beispiel für ein einfaches Multiprozessorsystem. Im folgenden Kapitel werden einige Hardware-Eigenschaften von Multiprozessor-Systemen untersucht.

5.4 Harware-Eigenschaften komplexer Systeme

5.4.1 Monoprocessing - Multiprocessing

Rechnersysteme sind dann überlastet, wenn sie ihre Aufgaben nicht mehr in der geforderten Zeit erledigen können. Um diesem Zustand abhelfen zu können, sind prinzipiell zwei Wege möglich:

1. Steigerung der CPU-Verarbeitungsgeschwindigkeit,
2. Multiprocessing.

Die Steigerung der CPU-Geschwindigkeit ist in den Fällen ein gangbarer Weg, wo

a) die momentan eingesetzte CPU nicht in Maximalgeschwindigkeit betrieben wird,

b) auf dem Markt eine andere CPU höherer Geschwindigkeit angeboten wird,

c) der Platzbedarf und der Preis keine so ausschlaggebende Rolle spielen, so daß eventuell ein Minicomputer eingesetzt werden kann.

Fall a) (nicht in Maximalgeschwindigkeit betriebene CPU) muß eventuell dann in Kauf genommen werden, wenn die eingesetzten Speicher für die CPU zu langsam sind und die CPU keine Wartezyklen einlegen kann, weil z.B. keine Schaltung auf der Baugruppe vorhanden ist, die die Wartezyklen erzeugt ("Wait-State-Generator"). Hier hilft nur ein Umrüsten des Rechners auf schnellere Speicher.

Fall b) (andere CPU höherer Geschwindigkeit) wird erwogen werden können, wenn beim Umstieg die vorhandene Unterstützung für Entwicklungen größtenteils weiterverwendet werden können, also z.B.

Entwicklungssysteme, vorhandene Software-Bibliotheken usw. So ist bei Neuentwicklungen der Umstieg von einem 8085 auf einen 8088 bei 8-Bit Rechnern oder von einem 8086 auf einen 80286 bei 16-Bit Rechnern durchaus denkbar.

Die Entscheidung für Fall c) (Minicomputer statt Microcomputer) ist auch für Neuentwicklungen so weitreichend und kostenträchtig, daß vorher mögliche Optimierungen des bereits vorhandenen Systems auf ihre Tauglichkeit untersucht werden sollten.

Die zweite Möglichkeit, die Verarbeitungsgeschwindigkeit eines Rechnersystems zu steigern, besteht darin, die anfallenden Aufgaben von mehereren Rechnern gleichzeitig erledigen zu lassen (Multiprocessing). Man unterscheidet zwischen

Funktionsverbund und Lastverbund.

Im Funktionsverbund sind verschiedene Aufgaben auf verschiedene, parallel arbeitende Rechner verteilt. Im Lastverbund teilen sich verschiedene Rechner die Bearbeitung derselben Aufgabe.

Ein Beispiel für Funktionsverbund ist die Bedienung von Plattenspeichern durch einen Rechner, und die Bedienung von Datenübertragungsstrecken durch einen anderen, parallel arbeitenden Rechner. Ein Beispiel für Lastverbund ist, wenn in einem System mehrere Datenübertragungsrechner vorhanden sind, die zwar alle mit identischer Hard- und Software bestückt sind, aber jeder für bestimmte Datenleitungen zuständig ist.

Betrachten wir als Beispiel folgende Konfiguration (Abb. 5-9):

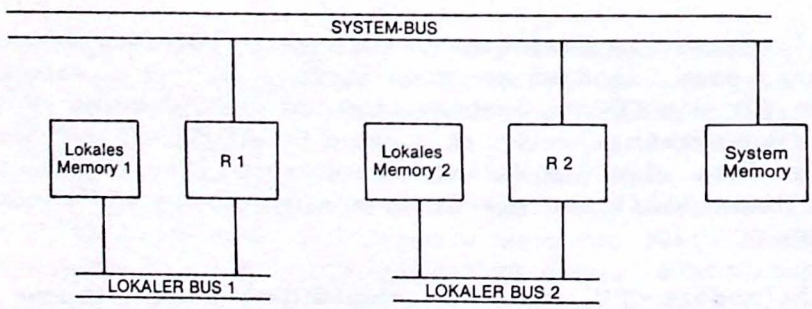

Abb. 5-9: Multiprozessor-System in fester Kopplung

Die Rechner 1 (R1) und Rechner 2 (R2) sind zu einem System in "fester Kopplung" zusammengefügt. "Feste Kopplung" bedeutet, daß beide Rechner direkt auf einen gemeinsamen Speicher, das System-Memory, zugreifen können. Um die Rechnerleistung eines Multiprozessor-Systems möglichst hoch zu halten, muß man dafür sorgen, daß nicht unnötig auf den gemeinsamen Systembus zugegriffen wird. Daher gibt man den einzelnen Rechnern oft noch zusätzlichen lokalen Speicher, der nur einem Rechner bekannt ist, und nur gemeinsame Datenbereiche liegen im Systemspeicher.

Eine häufig verwendete Variante ist ein sog. "Dual Port RAM", auf das zwei Rechner (quasi-)gleichzeitig zugreifen können, wobei aber ein- und dieselbe Speicherzelle für beide Rechner im allgemeinen auf verschiedenen Adreßbereichen liegt (Abb. 5-10).

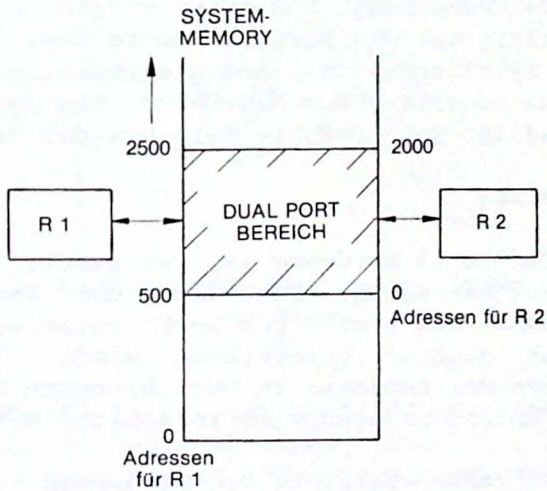

Abb. 5-10: System Memory mit Dual Port-Bereich

Hier hat R1 Zugriff auf das gesamte System Memory, einschließlich dem Dual Port Bereich. R2 kennt dagegen vom System Memory nur den Dual Port Bereich. Dual Port RAM Speicher ermöglichen Multiprocessing ohne zusätzliche Belastung des Systembuses, da Wartezeiten am Bus nur auftreten, wenn beide Partner gleichzeitig auf den Dual Port Bereich zugreifen. Anordnungen in "fester Kopplung" sind bei Rechnern, die nicht im selben Gehäuse untergebracht sind, oft unbequem, da der komplette Systembus mit seinen vielen

Leitungen verdrahtet werden muß, so daß auch hier gern auf "lose Kopplungen" z.B. mit seriellen Datenübertragungen zurückgegriffen wird. Bleiben wir jedoch vorerst bei der festen Kopplung der Rechner.

5.4.2 Multiprocessing - Feste Kopplung

In Abb. 5-9 können R1 und R2 gleichzeitig auf das System Memory zugreifen. Folgende Mechanismen müssen vorgeleistet sein, damit dieser Rechnerverbund funktioniert:

Hardware-Verriegelung

Eine geeignete Hardware-Logik muß dafür sorgen, daß nicht beide Rechner gleichzeitig auf den Speicher zugreifen. Dafür muß ein Prioritätsschema existieren, das bei gleichzeitigem Zugriffsversuch einem den Bus zuteilt ("Bus Arbiter"). Ein bekannter solcher Multiprozessor-Bus ist der erwähnte Multibus der Fa. Intel.

Software-Verriegelung

Wenn dieser Abschnitt im Hardware Kapitel steht, dann deswegen, weil trotz der softwaremäßigen Behandlung der Verriegelung der Hardware-Mechanismus des sog. "Bus Lock" vorgeleistet sein muß, was im folgenden genauer beschrieben wird. Die Behandlung gleichzeitig laufender Prozesse in verschiedenen Rechnern durch ein "verteiltes" Betriebssystem wird in Kapitel 6.4 erläutert.

Schreibt R1 in den Memory-Bereich, den R2 gerade liest, könnte R2 Daten lesen, die zusammengehören, aber erst teilweise von R1 geändert wurden. Um das zu vermeiden, muß R1 seine Datenstruktur "verriegeln". Das geschieht mit sog. "Flags", oft auch "Semaphore" genannt, die aber nicht mit den Semaphoren verwechselt werden dürfen, die von einem Betriebssystem verwaltet werden. (s. 2.5.3) Das "Flag" ist eine Bool'sche Variable im interessierenden Bereich des Systemspeichers, also z.B. ein Bit, das anzeigt, ob der Speicherbereich frei ist oder gerade einer der an den Bus angeschlossenen Rechner darauf arbeitet. Eine "1" heißt, daß der Bereich belegt ist, eine "0", daß er frei ist.

Die Anforderung des Speicherbereiches ist dann kritisch, wenn zwei Rechner zur gleichen Zeit darauf zugreifen wollen (Abb. 5-11).

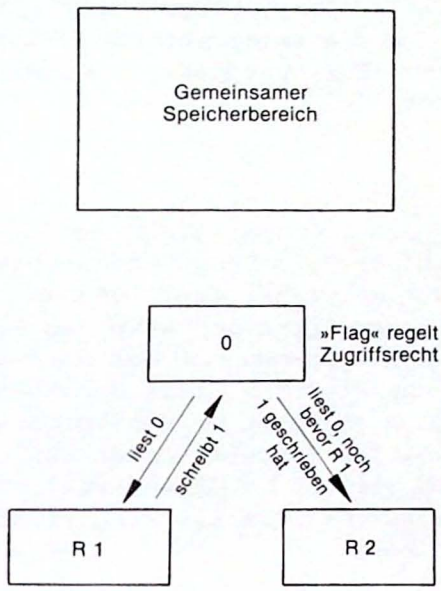

Abb. 5-11: Konflikt bei teilbarer "Teste und Setze" Funktion

Das "Flag" in Abb. 5-11 hat den Wert "0", der Speicherbereich ist also frei. Falls R1 das Flag liest, sieht er, daß der Speicher frei ist und arbeitet darauf. Um anzuzeigen, daß er den Speicher benutzt, setzt R1 das Flag auf "1". Ebenso geht es R2, wenn er das Flag liest, noch bevor es R1 gesetzt hat. Um diesen Konfliktfall zu vermeiden, muß ein Rechner das Flag lesen und setzen können (Teste-und-Setze Funktion), ohne den Bus abgeben zu müssen. Dazu reicht z.B. beim 8086 der XCHG-Befehl, der es erlaubt, ein Register mit einer Speicherzelle (also den "Besetzt"-Wert mit dem aktuellen Inhalt des Flags) auszutauschen, nicht aus. Der einzelne XCHG-Befehl nämlich besteht wieder aus einzelnen sog. Buszyklen, zwischen denen ein anderer Rechner zum Zug kommen kann.

Der Bus muß also zumindest für die Dauer der "Teste-und-Setze" Funktion verriegelt werden können, in unserem Beispiel also für die Dauer des XCHG-Befehls. In einem 8086 System, wo der Prozessor als "Master" über einen Bus Arbiter 8289 an den Multibus angeschlossen ist, wird der Bus für die Dauer eines Befehls, dem im Programm ein "LOCK" Prefix vorangestellt ist, verriegelt. Die Voraussetzung ist, daß die entsprechenden Signale zwischen 8086, 8289 und dem System Bus verdrahtet sind. Durch die 8086-Assembler-Befehlsfolge

```
     MOV  AL,1
LOCK XCHG AL, FLAG
```

wird die "Teste-und-Setze" -Funktion durchgeführt. Hier wird zuerst der Akkumulator mit 1 geladen, und im nächsten Befehl der Inhalt des Akkus mit dem aktuellen Wert des Flags ausgetauscht, wobei der "LOCK"-Prefix zusammen mit der Hardware die Exklusivität des Zugriffs sicherstellt. Liest der Rechner, daß das Flag auf "besetzt" zeigt, wiederholt er den Vorgang solange, bis er "frei" vorfindet, d.h. der Partnerrechner den gemeinsamen Bereich freigegeben hat. Man spricht in diesem Fall vom sog. "busy waiting" (beschäftigtes Warten), da das Flag ständig abgefragt wird, solange es "besetzt" zeigt.

Ist die hardwaremäßige "LOCK"-Möglichkeit nicht gegeben, kann man auch anders vorgehen, wenn das Mehrrechnersystem nur aus zwei Partnern besteht. Jetzt bekommt jeder der beiden Rechner ein Flag, was vor jedem Zugriffsversuch auf einen gemeinsamen Bereich gesetzt wird. Nach dem Setzen der eigenen Anforderung wird das Partner-Flag solange abgefragt, bis es "frei" zeigt. Diese Methode eignet sich z.B. für sog. "Dual-Port-RAM" Konfigurationen (s. oben), bei denen einer oder beide Rechner keinen Zugriff zum "LOCK"-Mechanismus der Hardware haben.

5.4.3 Multiprocessing - Lose Kopplung

Auf "lose Kopplungen" muß man zurückgreifen, wenn aus einem der oben erwähnten Gründe die "feste Kopplung" nicht möglich oder gewünscht ist. Ein häufiges Beispiel ist, daß Datenmengen über größere Strecken transportiert werden müssen. Hier muß man über serielle oder parallele Datenleitungen, die für größere Entfer-

nungen ausgelegt sind, gehen. Beispiele für serielle Datenleitungen sind die V.24 Schnittstelle oder die X.21 Schnittstelle, über die oft ein HDLC Protokoll abgewickelt wird. Beispiele für parallele Datenleitungen sind der sog. IEC-Bus oder der CAMAC-Bus. Im Betriebssystem muß ein Transportsystem (s. 1.1.4) für den geordneten Datentransfer sorgen.

Die folgenden Abschnitte zeigen Anwendungsbeispiele der erläuterten Prinzipien.

5.4.4 EWSD - Ein Beispiel für ein Multiprozessorsystem

Es wird an einem Beispiel gezeigt, welche Leistungsfähigkeit in einem komplexen Multiprocessor-System, das aus Mikrocomputern aufgebaut ist, steckt. Es handelt sich um das digitale Fernsprechsystem EWSD (Siemens AG), das von der Deutschen Bundespost im öffentlichen Telefonverkehr eingesetzt wird (Abb. 5-12, 5-13). Die Abbildungen stammen aus der Zeitschrift 'telcom report 4(1981)' und sind wiedergegeben mit freundlicher Genehmigung der Siemens AG, München.

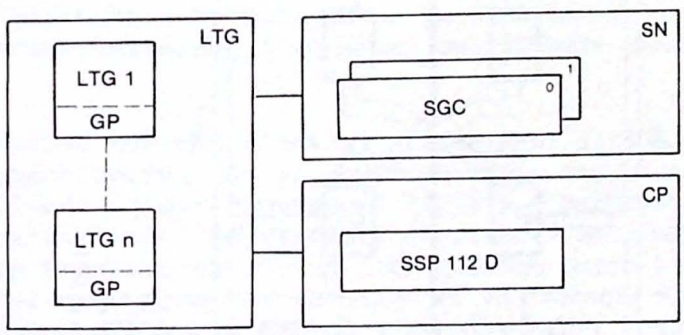

Abb. 5-12: EWSD-Vermittlungsstelle

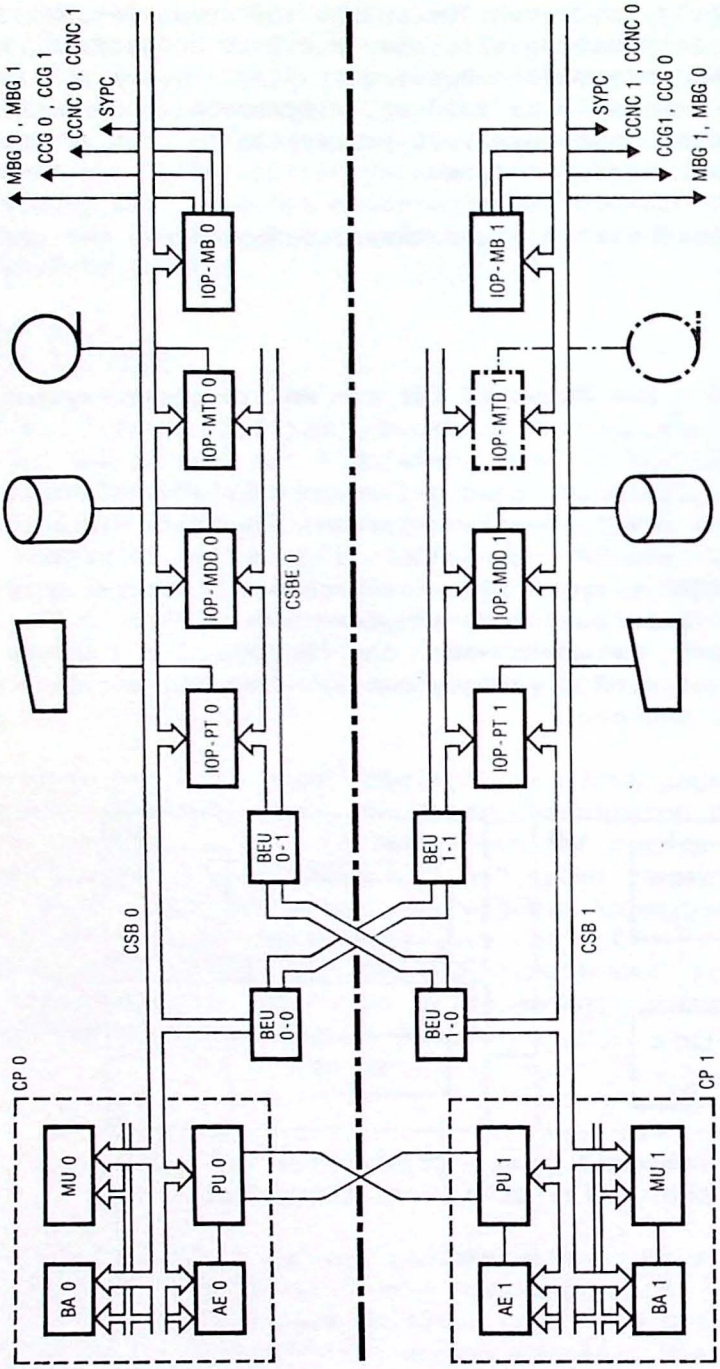

Abb. 5-13: Architektur des SSP 112

Die Abb. 5-12 gibt eine grobe Übersicht über das Steuerungskonzept der EWSD-Vermittlungsstelle. Die Signalisierung vom und zum Teilnehmer (z.B. Wählen, Rufe) wird über die LTGs (Line Trunk Groups) abgewickelt. Die Zahl der LTGs ist variabel. An jede LTG können bis zu 256 Teilnehmer angeschlossen werden, deren Signalisierung von einer Mikroprozessorsteuerung, dem GP (Group Processor) verarbeitet wird.

Die GPs sind über serielle Datenleitungen, über die ein HDLC Protokoll abgewickelt wird, mit dem CP (Central Processor) verbunden. Der CP ist den GPs übergeordnet (Steuerungshierarchie). Der dritte Komplex SN (Switching Network = Koppelnetz) nimmt die Verbindung der Sprachwege je zweier oder mehrerer (Konferenz) Teilnehmer vor. Das SN wird vom SGC (Switching Group Control) gesteuert, der ebenfalls dem CP untergeordnet ist.

Die Abb. 5-13 zeigt die Architektur des Central Processors CP, genannt SSP 112 (SSP = Siemens Switching Processor). Die Indizierung mit 0 und 1 bezeichnet zwei identische Hälften, von denen jeweils eine aktiv ist, während die zweite in Warteposition steht. Das System ist auf Basis des Mikroprocessors 8086 (Intel) aufgebaut. Der zentrale Teil (CP0, CP1) besteht aus PU (Processing Unit), der Speichereinheit, dem Buszuteiler BA (Bus Arbiter) und der Adreßerweiterung AE (Address Extension). Die Adreßerweiterung ist nötig, da für den Anwendungszweck der vom 8086 unmittelbar adressierbare Speicherraum von 1 MByte nicht ausreicht. Mit der Adreßerweiterung können maximal 8 MByte adressiert werden.

An den zentralen Bus CSB (Central System Bus) sind aber noch mehr Rechner angeschlossen. Es sind verschiedene Ein-/Ausgabe Prozessoren IOP (Input-Output-Processor) für z.B. Bandstationen, Magnetplattengeräte oder Bedienblattschreiber. Außerdem sind noch Ein-/Ausgabe Prozessoren IOP-MB (IOP-Message Buffer) zu den Nachrichtenverteilergruppen MBG (Message Buffer Group) vorhanden, die den Nachrichtenverkehr zu den LTGs kanalisieren. Die IOPs besitzen auch lokalen Speicher, was aus Abb 5-13. nicht hervorgeht.

Die Verbindung zwischen PU0 und PU1 bedeutet, daß die einzelnen Rechner über diese Leitungen wissen, welche Hälfte gerade aktiv ist. Fällt ein CP aus, sind dessen IOPs über die Buserweiterungseinheit BEU noch von der zweiten Hälfte ansprechbar.

Dieses Steuerungskonzept zeigt verschiedene der oben erwähnten Prinzipien:

Multiprozessorsystem: LTGs, CPs und innerhalb der CPs die PU und IOPs bilden ein Multiprozessorsystem, in dem zur selben Zeit verschiedene Aufgaben abgewickelt werden.

Lose Kopplung: LTGs und CPs kommunizieren über größere Entfernungen über serielle Datenleitungen mit einem gesicherten Datenübertragungsverfahren (HDLC).

Feste Kopplung: PU und IOPs des Zentralrechners sind an einen gemeinsamen Systembus angeschlossen für schnellen parallelen Datenaustausch über kurze Strecken.

Funktionsverbund: LTGs, CP und innerhalb des CP die PU und die IOPs arbeiten zur gleichen Zeit an verschiedenen Funktionen.

Lastverbund: die LTGs arbeiten alle an der gleichen Aufgabe und teilen sich die "Last", für viele Teilnehmer dieselben vorverarbeitenden Funktionen vorzunehmen.

Die Leistungsfähigkeit dieser komplett aus Mikrocomputern aufgebauten Steuerung ist beachtlich: Es können an ein EWSD System mit dieser Rechnerstruktur bis zu 12000 Teilnehmer angeschlossen werden; die Anlage ist spezifiziert für 60000 BHCA (Busy Hour Call Attempts, Verbindungswünsche in der Hauptverkehrsstunde). Da die eingesetzten Prozessoren (8086) bezüglich ihres Adressierungsvolumens über die Grenze von 1 MByte ausgereizt sind, ist zu erwarten, daß für weitere, noch leistungsfähigere Anlagen (100000 Teilnehmer) leistungsfähigere Prozessoren eingesetzt werden, sobald diese verfügbar sind.

5.4.5 Redundanz

Die waagrechte Strichlierung in Abb. 5-13 trennt zwei identische Rechnerhälften. Von den beiden Hälften ist die eine aktiv, die andere in Wartestellung und bereit, im Fall eines Fehlers in der aktiven Hälfte, den Betrieb zu übernehmen.

Dieses Prinzip wird bei Systemen angewendet, die eine hohe Zuverlässigkeit aufweisen müssen, wobei bei auftretenden Fehlern keinesfalls sofort das ganze System ausfallen darf (fehlertolerante Systeme). Die Umschaltung im Fehlerfall geht beispielsweise so vor sich: In Abb. 5-13 sind beide PUs über zwei Leitungen verbunden. Eine der beiden Leitungen pro PU geht aus von einem Ausgangsport, der einen Wert für den Aktiv-Zustand und einen Wert für den Standby-Zustand einnehmen kann. Die zweite der beiden Leitungen verbindet diesen Ausgangsport der Gegenseite mit einem Eingangsport, wo der Zustand der zweiten Hälfte abgefragt wird. Man sagt, die Leitungen sind "ausgekreuzt".

Passiert nun in der aktiven Hälfte ein nichtbehebbarer Fehler, z.B. Ansprechen des Watchdog (s. 5.2.3), so merkt das die Standby-Hälfte, da diese über ihren Eingangsport den Zustand der anderen Hälfte ständig abfragt. Daraufhin übernimmt sie den Betrieb. Nachdem die defekte Hälfte repariert wurde, fungiert sie als Standby-Hälfte.

Aus Abb. 5-13 ist zu entnehmen, daß sämtliche Teile des Zentralprozessors, der PU, gedoppelt sind. Die peripheren Rechner, die die LTGs steuern (Abb. 5-12) sind nicht gedoppelt. Diese bilden sog. Ausfalleinheiten. Das heißt z.B. beim EWSD-System, daß bei Ausfall einer LTG maximal 256 Teilnehmer gestört sind, die restlichen nicht. Damit Einheiten eines Rechnersystems als Ausfalleinheiten arbeiten können, müssen sie, ohne Beeinträchtigung anderer Teile, reparierbar sein. Das heißt insbesondere, daß sie eigene Stromversorgungen haben müssen.

6. Systementwurf in Hinblick auf die Umgebung eines Realzeit-Multitasking-Betriebssystems Software-Aspekte

Dieses Kapitel behandelt einige Probleme des Software-Systementwurfes für Realzeitsysteme aus der Sicht des Anwenders. Unter "Anwender" soll hier etwa ein Personenkreis verstanden werden, der ein Realzeit-Betriebssystem zur Lösung einer Aufgabe einsetzen möchte. Es werden also nicht in erster Linie interne Probleme eines Betriebssystems behandelt, sondern solche Probleme, die gelöst werden müssen, damit man zu einer befriedigenden und zuverlässigen Funktion der Software gelangt, die unter der Kontrolle des Betriebssystems abläuft.

6.1 Allgemeine Kriterien zur Software-Systemstruktur

Die Struktur eines Rechnersystems, insbesondere die Software-Struktur, sollte "Top-Down", d.h. von oben nach unten gefunden werden. Man spricht von "schrittweiser Verfeinerung". Übliche Stationen eines Top-Down Systementwurfes sind:

1. Pflichtenheft: Vorgaben vom Auftraggeber.

2. Rahmenspezifikation: Allgemeine Systembeschreibung, sowohl Hardware als auch Software. Spezifikation der Leistungsmerkmale des Gesamtsystems. Pauschale Beschreibung der einzelnen Software-Komponenten. Die grundlegenden Voraussetzungen zur Software-Entwicklung (Programmiersprache, Betriebssystem, Organisation des Software-Supports, Abschätzung über den erwarteten Entwicklungsaufwand) stehen fest.

3. Grobspezifikation: Beschreibung der einzelnen Software-Komplexe, ihre Aufteilung in Prozesse, sowie die logische Struktur der Schnittstellen. Die Software-Schnittstellen sind vollständig aufgelistet und vom Inhalt her beschrieben, ohne jedoch vom Format her schon bis auf "Bit-Ebene" festzuliegen.

4. Detailspezifikation: Detaillierte Beschreibung der Schnittstellenformate. Beschreibung der Prozeßorganisation z.B.in Form von Zustandsdiagrammen (s.7.2.2). Arbeitsgrundlage zur Programmierung ("Codierung") und Bezugsdokument für spätere Fehlersuche.

5. Codierung: Erstellung des Programms nach den Regeln der strukturierten Programmierung auf der Basis der Detailspezifikation.

6.2 Realisierung der Software-Struktur

Einige Stationen des Software-Systementwurfes, die sich besonders auf den Einsatz von Realzeit-Multitasking-Betriebssystemen beziehen, werden näher erläutert.

6.2.1 Wahl des Betriebssystems

Die hier interessierende Frage des Software-Systementwurfes ist, ob bei einem gegebenen Problem ein Realzeit-Multitasking-Betriebssystem eingesetzt werden soll. Diese Frage ist wiederum gliedern:

1. Kann ein kommerzielles Betriebssystem eingesetzt werden?

2. Falls nein, kann dann überhaupt eines eingesetzt werden?

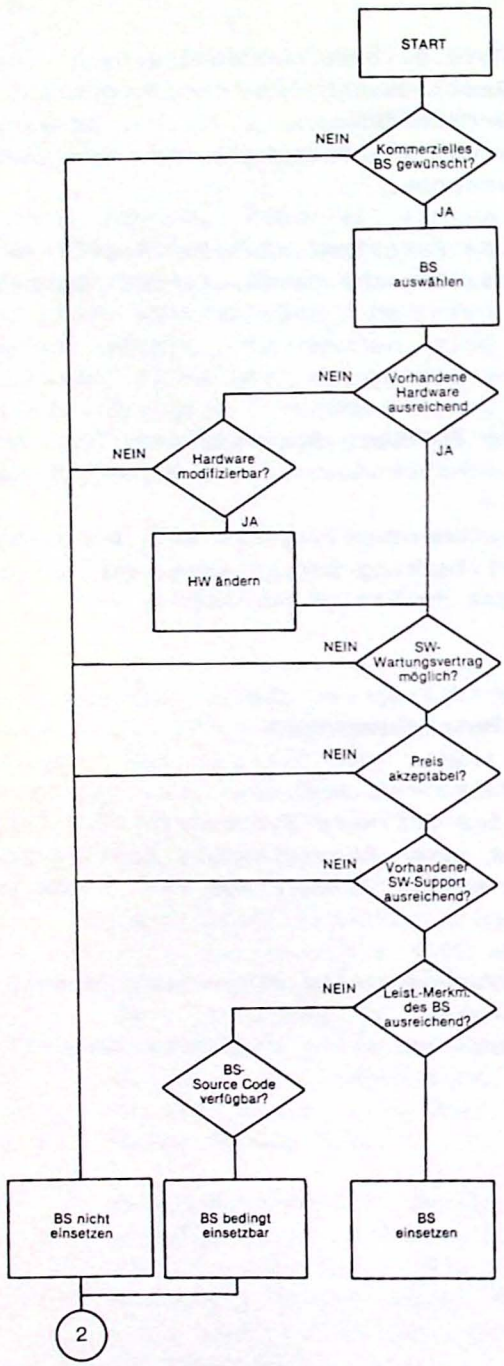

Abb. 6-1: Kommerzielles BS einsetzbar?

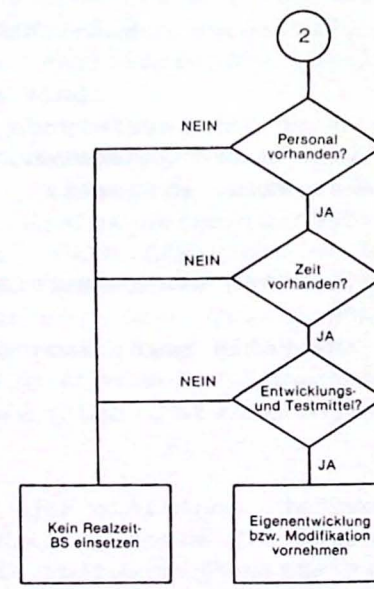

Abb. 6-2: Eigenentwicklung eines BS durchführbar?

Die Abb. 6-1 bildet ein Entscheidungsdiagramm zur ersten Frage, Abb. 6-2 einen zur zweiten Frage. Der erste Entscheidungsbaum bietet drei Ausgänge:

1. Das betrachtete Betriebssystem ist für die Anwendung einzusetzen.

2. Das Betriebssystem ist bedingt einsetzbar, d.h. es sind Modifikationen durchzuführen. Dazu muß der Quell-Code verfügbar sein. Unter diesem Aspekt ist die Frage nach den Kosten natürlich neu zu stellen. In diesem Fall sind anschließend die gleichen Fragen zu stellen wie im Fall einer Eigenentwicklung.

3. Das Betriebssystem ist nicht einsetzbar. Landet man bei diesem Ausgang, sollte man sich nach einem anderen Betriebssystem umsehen oder eventuell eine Eigenentwicklung in Betracht ziehen, was in den zweiten Entscheidungsbaum führt (Abb. 6-2).

Der zweite Entscheidungsbaum ist zu durchlaufen, wenn man Eigenentwicklung eines Betriebssystems oder die Modifikation eines vorhandenen Source-Codes vornehmen will. Es gibt zwei Ausgänge:

1. Die Eigenentwicklung bzw. Modifkation eines vorhandenen Betriebssystems ist möglich.

2. Es kann unter den gegebenen Voraussetzungen kein Realzeit-Betriebssystem eingesetzt werden. In diesem Fall ist zu fragen, ob für die Komplexität der gestellten Aufgabe wirklich ein Realzeit-Multitasking-Betriebssystem erfordert.

In den Beispielen der folgenden Abschnitte soll auch versucht werden, ein Gefühl zu vermitteln, wie ein Problem in verschiedene Prozesse aufgeteilt werden soll. Kommt man zu dem Entschluß, daß ein einziger Prozeß ausreichen würde, braucht man auch kein Multitasking-Betriebssystem. In allen anderen Fällen sollte man jedoch nicht auf den Einsatz eines solchen verzichten, um sich von Anfang an die Flexibilität des Entwurfes zu sichern.

6.2.2 Analyse der Aufgabenstellung - Aufteilung in Prozesse

Hat man sich für den Einsatz eines Multitasking-Betriebssystems entschieden, muß die gestellte Aufgabe so analysiert werden, daß eine sinnvolle Aufteilung in einzelne Prozesse gefunden wird. Das ist der kritischste Teil des Software-Entwurfes. Aus der erreichten Lösung wird oft erst später sichtbar, wenn Änderungen anstehen, ob die erstellte Software den häufig benutzten Schlagworten wie "Modularisierung", "Änderungsfreundlichkeit", "gute Wartbarkeit" usw. gerecht wird. Diese Schlagworte werden oft allein auf die Verwendung einer höheren Programmiersprache und den Einsatz eines Betriebssystems bezogen. Die einzelnen Programmoduln sind dann vielleicht insofern modular, daß sie abgeschlossene Übersetzungseinheiten bilden. Das nützt jedoch recht wenig, wenn bei Änderungen in einem Modul an vielen anderen Moduln des Programmsystems ebenfalls geändert werden muß. An dieser Stelle des Software-Entwurfes lohnt es sich, gründlichst über mögliche spätere Erweiterungen des Systems nachzudenken. Ebenso sollte man versuchen, mögliche Fehler- und Konfliktfälle vorher im Gedankenexperiment ausgiebig zu analysieren.

6.2.2.1 Ein Beispiel

Am ehesten wird das eben Vorgestellte an einem Beispiel deutlich. Das soll einerseits zeigen, wie ein Programmsystem sinnvoll in einzelne Prozesse aufgeteilt wird, andererseits die Variabilität und Anpaßbarkeit des Prozeßkonzeptes gegenüber einem konventionellen, linearen Programmablauf herausstellen.

Betrachten wir die Steuerung der Wettkampfanzeige in einem Sportstadion, z.B bei einem Schwimmwettkampf. Die Anzeigetafel soll, gesteuert von einer Eingabetastatur, verschiedene Informationen für die Zuschauer bieten, z.B. sollen die Namen der Sportler in alphabetischer Reihenfolge aufgelistet werden können.

Das wesentliche aber ist die Anzeige von aktueller Information während des Wettkampfes, also die Anzeige der Stoppuhr oder der erreichten Zwischenzeiten der einzelnen Sportler. Außerdem sollte unmittelbar nach Abschluß des Wettkampfes das Endresultat fest-

stehen mit einer Genauigkeit von 1/100 sec oder genauer. Der konkrete Ablauf könnte folgendermaßen aussehen:

Auf ein Vorbereitungssignal hin betreten die Schwimmer die Startblöcke, auf den Startschuß hin beginnen sie das Rennen. Bei jeder Wende setzen sie durch Anstoßen der Bahnbegrenzung eine Zeitmarke, und das über die vorher festgelgte Distanz.

Wir wollen dieses Beispiel auf zwei verschiedene Arten analysieren. Der erste Vorschlag benutzt einen konventionellen, linearen Programmablauf, der ohne Prozeßkonzept und Realzeit-Betriebssystem arbeitet. Der zweite Vorschlag benutzt das Prozeßkonzept und ist mit einem Realzeit-Betriebssystem zu verwirklichen. Daraus werden die Vorteile der Benutzung des Prozeßkonzeptes klar.

Jetzt also zuerst der konventionelle Realisierungsvorschlag:

Die Zergliederung der Aufgabe könnten wir uns bei einem linearen Programmablauf so vorstellen:

1. Vorbereitende Eingaben an der Bedientastatur, z.B. Distanz, Zahl der belegten Bahnen usw.

2. Nach dem Betreten der Startblöcke durch alle Teilnehmer Startfreigabe durch den Bediener.

3. Warten auf Startschuß oder auf Fehlstart. Nach Fehlstart zurück zu 2.

4. Start geglückt: Uhr wird durch den Startschuß gestartet, Freigabe der Bahnbegrenzungsbewertung.

5. Aufnahme der Bahnbegrenzungsanschläge z.B. indem alle Bahnen in genügend kurzen Zeitabständen abgefragt werden, oder indem jeder Bahnbegrenzung ein eigener Interrupt zugeordnet wird. Dabei wird aus der Zahl der aufgenommenen Anschläge für jeden Teilnehmer entschieden, ob der Anschlag als Zwischenzeit oder als Endzeit oder gar nicht an der Tafel angezeigt wird.

6. Alle 1/10 sec nach "Start geglückt" Interrupt zur Anzeige der gerade laufenden Zeit.

Hier sei noch eine Betrachtung über die Zeitaufnahme angestellt:

Falls die Zeitauflösung 1/1000 sec betragen soll, ist diese Genauigkeit durch eine softwaremäßige Lösung nur schwer zu erreichen, da (bei 10 Bahnen) bei zyklischer Abfrage von 10 Startblöcken und 20 Bahnanschlagtafeln insgesamt 30 Sensoren abzufragen sind. Man hat also bei der geforderten Zeitauflösung nur

1ms / 30 = 30 us / Sensor

zur Verfügung zur Erfassung der Änderung des Sensors und das Notieren der dazugehörigen Zeit der Stoppuhr. Da nur der Zeitpunkt einer Änderung auf 1 msec genau festgehalten werden muß, die Änderungen eines Sensors jedoch nicht so schnell aufeinander folgen können, ist hier eine Hardwarelösung für die Zeitmessung angebracht. Diese könnte so aussehen, daß jede Änderung eines Sensors mit der dazugehörigen Zeit der Stoppuhr in einen Pufferspeicher eingetragen wird, der dann vom Rechner in aller Ruhe, etwa alle 20 msec ausgelesen werden kann. Dieser Pufferspeicher ist dann die Schnittstelle zur Software. Bei weniger strengen Zeitforderungen, z.B. einer geforderten Auflösung von 1/100 sec ist die Zeitmessung auch per Software zu erreichen.

Nach diesem Ausflug zurück zu unserer Software-Aufgabenteilung.

Für den betrachteten linearen Programmablauf sind folgende Moduln denkbar:

VORBEREITUNG:
Abfrage der Namen der Teilnehmer im Dialog, durch spezielles Eingabezeichen Anzeige auf der Tafel (mit Modul ANZEIGE). Weitere besondere Eingabe für die Startfreigabe.

ERWARTE_START:
Periodisches Abfragen der Startpistole bzw. einer Einrichtung, die einen Frühstart signalisiert. Nach einem eventuellen Fehlstart zurück in VORBEREITUNG, wobei dann nur noch die Startfreigabe erfolgen muß.

BEWERTE_BAHN:
Mit dem geglückten Startschhuß werden die Bahnbegrenzungsanschläge freigegeben. Damit startet ein Programm, das in periodischer Folge entweder die Bahnbegrenzungen selber abfragt, oder den Pufferspeicher, in den eine Hardwarelogik die Anschläge der

Bahnmarken mit den zugehörigen Zeiten einträgt, je nach geforderter Zeitauflösung. Falls Zwischenzeit, mit Modul ANZEIGE zur Anzeige bringen, falls Endzeit, ebenfalls anzeigen und Position markieren.

ANZEIGE:
Anzeige von Text auf der Stadiontafel. In diesem Programm muß der 1/10 sec-Interrupt (Zeitanzeige während des Rennens) gesperrt werden, da er ebenfalls ANZEIGE benutzt.

INTERRUPT-PROGRAMME:
Zeitinterrupt, der den BEWERTE_BAHN-Scanner anstößt, 1/10 sec-Interrupt zur Anzeige der laufenden Zeit, evtl. weiter Interrupts zur Ein-/und Ausgabe am Bedienterminal.

RAHMEN:
Dieser Modul bildet die Ablaufsteuerung für das Programmsystem. Er bestimmt die zeitliche Abfolge, in dem die anderen Moduln aufgerufen werden.

Ohne diesen Realisierungsvorschlag weiter zu detaillieren, ist sehr wohl denkbar, daß die Entwickler für ihr Programmsystem die "modulare Programmierung" beanspruchen. Es gilt für die einzelnen Moduln:

> Präzis definierte Schnittstellen,
> Überschaubarkeit,
> von einer Person gut beherrschbar.

Trotzdem hat die gewählte Lösung vom Ansatz her gravierende Mängel, wie sich bei näherem Hinsehen herausstellt:

1. Nicht alle Sensormeldungen können gleich behandelt werden. In der Startphase sind Startpistole und Absprungerkennung sowie die Frühstarterkennung zu bewerten, in der Rennphase nur noch die Bahnbegrenzungsanschläge. Für Staffelwettbewerbe wäre eine Verkopplung von Bahnbegrenzung und Absprungerkennung zur Erkennung der korrekten Übergabe zum nächsten Staffelschwimmer wünschenswert. Im vorliegenden Entwurf ist das nur schwer möglich.

2. Soll ein ähnliches Programm für andere Sportarten verwendet werden, erleidet man völlig Schiffbruch, d.h. so gut wie das komplette Programm ist umzustellen. Zum Beispiel die Disziplin

Skilanglauf verlangt, daß die einzelnen Teilnehmer nacheinander starten, wobei nicht wie beim Schwimmen jeder seinen eigenen "Startblock" hat, sondern alle durch dieselbe Zeitschranke starten.

3. Der Modul ANZEIGE wird von mehreren anderen Moduln benutzt, insbesondere vom 10 msec Interruptprogramm. Um Konfliktfälle zu vermeiden, muß während ANZEIGE läuft, dieser Interrupt gesperrt werden. Andere Unterbrechungen sollten jedoch zugelassen bleiben, z.B. Eingaben von der Bedientastatur. ANZEIGE braucht also detaillierte Kenntnis über das Unterbrechungsverhalten der Maschine.

Man sieht also, daß die oben erwähnten drei Punkte (präzise Schnittstellen, Überschaubarkeit, Beherrschbarkeit) wohl für die einzelnen Moduln gelten mögen, nicht jedoch für das Gesamtsystem. Es sollte durchaus möglich sein, Änderungen an einem Programmsystem vorzunehmen, ohne alles neu entwickeln zu müssen. Denn Änderungen an Programmsystemen werden oft erst notwendig, wenn die Entwicklung des Systems schon sehr fortgeschritten, wenn nicht abgeschlossen ist (z.B. Forderung neuer Leistungsmerkmale durch den Kunden). Es ist bei den steigenden Kosten der Software im Verhältnis zur Hardware notwendig, neue Anforderungen an ein System ökonomisch entwickeln zu können.

Will man also bei der Lösung der erwähnten Probleme, beispielsweise von Punkt 2, nicht wieder von vorne anfangen, muß man sich überlegen, wie man den Moduln Funktionen so zuteilen kann, daß nicht bei relativ naheliegenden, trotzdem drastischen Änderungen das ganze Programmsystem wie ein Kartenhaus zusammenfällt. Die Verzahnung der einzelnen Programmteile, die keine isolierten, auf wenige Stellen begrenzten Änderungen erlaubt, müsste gelockert werden. Wie man an diesem Beispiel sieht, hilft der konventionelle Begriff von "Modularität" hier nicht weiter.

Das Entscheidende ist die Entkopplung des logischen Inhalts der einzelnen Moduln untereinander. Das bedeutet, daß einzelne Programmteile Funktionen ausführen, ohne daß sich andere um die Realisierung dieser Funktionen kümmern müssen. In unserem Beispiel heißt das für den Modul ERWARTE_START z.B., daß er nicht selber die Startblöcke und die Startpistole abfragen sollte; viel besser wäre es, wenn er "Nachricht" bekäme, ob der Start geglückt oder mißglückt ist.

Ähnlich beim Modul ANZEIGE. Es ist nicht gut und wird mit zunehmender Komplexität immer unübersichtlicher, wenn sich Moduln andere Programme wie ANZEIGE teilen, die unterbrechungsempfindlich sind.

Es ist durchaus vorstellbar, daß weitere Abhängigkeiten und Verriegelungen in die einzelnen Programmteile gebracht werden müssen, um Fehlschriften auf der Anzeigetafel zu vermeiden. Viel besser wäre es, einen eigenständigen Modul ANZEIGE zu haben, an den jeder seinen Text abgibt, wobei ANZEIGE völlig selbständig dafür sorgt, daß keine Konfliktfälle auftreten. Und falls doch Fehlschriften vorkommen, muß nur im Modul ANZEIGE gesucht werden und nicht auch noch in allen Moduln, die ANZEIGE benutzen.

Diese Probleme können bei Anwendung des Prozeßkonzeptes viel leichter vermieden werden, da hier die Schnittstellen nach außen ("Meldungen"), viel einfacher nach rein funktionellen Gesichtspunkten entworfen werden können, wobei der innere Programmaufbau, hier jetzt Prozeßaufbau, dem Außenstehenden verborgen bleibt.

Analysieren wir also nun das Beispiel mit Hilfe des Prozeßkonzeptes und legen für unser Programmsystem Funktionen, d.h. Prozesse fest (Abb. 6-3). An dieser Stelle werden wir etwas über den allgemeinen Aufbau, nicht über Details der Realisierung sagen. Das Beispiel wird jedoch später (s. z.B. 7.2) noch benutzt werden.

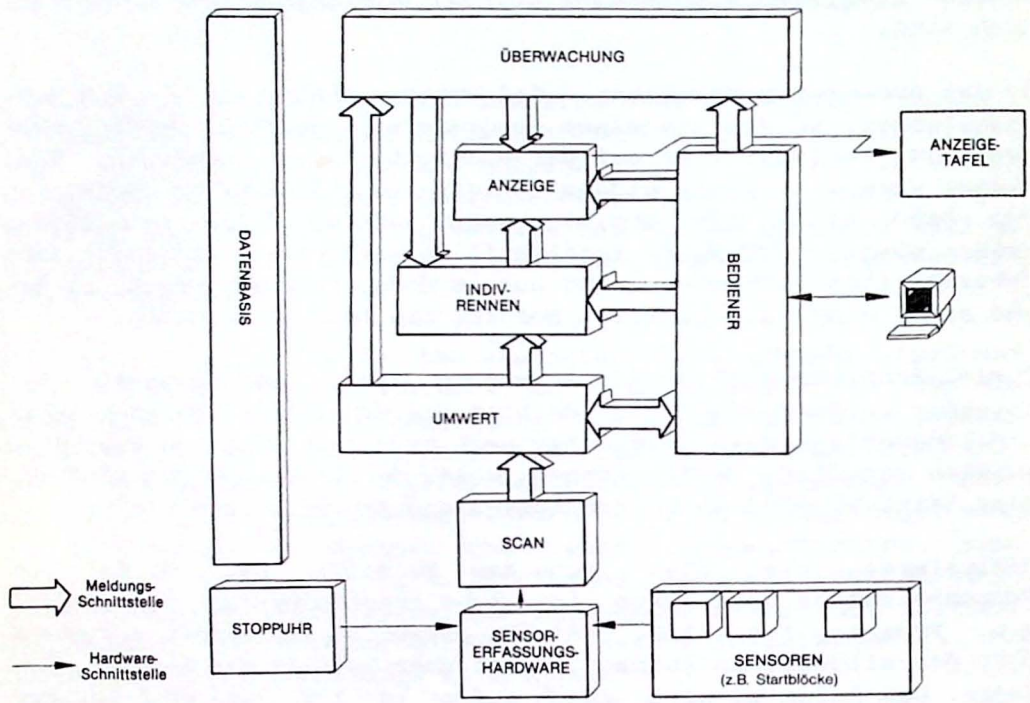

Abb. 6-3: Aufteilung des Problems in Prozesse

1. SCAN

Erkennung von Änderungen der Sensoren. Weitermeldung an UMWERT. Es wird bewußt von "Sensoren" und nicht von "Startblock" oder "Bahnanschlag" gesprochen, da SCAN keine Kenntnis über die logischen Eigenschaften der Sensoren hat. Beim Übergang auf andere Sportarten könnten die Sensoren z.B. Lichtschranken sein. SCAN braucht daher beim Übergang auf andere Sportarten nicht geändert zu werden, wenn ihm Anreize in derselben Form, z.B dem erwähnten Pufferspeicher angeboten werden.

2. UMWERT

Umwertung der von SCAN gemeldeten Anreize zu ihrem logischen Inhalt, am besten über Tabellenzugriffe. UMWERT hat Schnittstellen zu INDIV_RENNEN, an den die Sensoränderungen nun mit ihrem logischen Gehalt weitergegeben werden, etwa als Meldung BAHNANSCHLAG oder STARTBLOCK, natürlich mit Startnummer und Stoppuhrzeit als Parameter. UMWERT hat Schnittstellen in beiden Richtungen (SEND und RECEIVE) zu BEDIENER. Diese Schnittstelle ist etwa beim Skilanglauf wichtig, wo zum Zeitpunkt des Ereignisses eines Sensors (z.B. einer Zwischenzeit-Lichtschranke) die Startnummer von einem Streckenposten gemeldet wird. UMWERT kann dann über BEDIENER am Bedienterminal die Startnummer abfragen, vom Bedienpersonal eingeben lassen und danach an INDIV_RENNEN weitermelden.

Um die Zuordnung der Sensoren zu ihrem logischen Gehalt so flexibel wie möglich zu halten, sollte jedem Sensor eine Beschreibungstafel ("Deskriptor") zugeordnet sein, der von Sportart zu Sportart, oder sogar wie beim eben erwähnten Beispiel erst während des Rennens zugewiesen wird. Unterscheidungsmerkmale könnten etwa sein: Startnummer, Startpistole, Zwischenzeit, Startblock, zurückgelegter Weg usw. UMWERT kann Schnittstellen zu UEBERWACHUNG haben für Ereignisse, die alle Teilnehmer betreffen, z.B. wenn ein genereller Fehlstart passiert.

Es ist sogar denkbar, daß erreichte Zwischenzeiten von Hand eingegeben werden, vom Eingabeprozeß BEDIENER an UMWERT gemeldet werden und von dort an INDIV_RENNEN weitergemeldet werden, als wäre es ein Sensoranreiz gewesen. Man denke an Lichtschranken, die von mehreren gleichzeitig durchbrochen werden.

Im Prozeß UMWERT stehen die meisten Informationen über die individuelle Sportart. Er wird also von Fall zu Fall zu ändern sein.

Man kann hier konsequenterweise noch einen Schritt weitergehen und alle Daten, auf die von mehreren Prozessen zugegriffen wird oder die bei der Konfiguration des Systems geändert werden müssen, in einer sog. "Datenbasis" ablegen. Dann wäre im beschriebenen Fall nicht UMWERT zu ändern, sondern die Datenbasis mit einem neuen Satz von Sensor-Deskriptoren zu versehen. Auf die Probleme, die bei der Benutzung einer Datenbasis entstehen, wird in Abschnitt 6.3 eingegangen.

3. INDIV_RENNEN

Er heißt INDIV_, weil er individuell nach Startnummern organisiert ist. Eigentlich gehört zu jeder Startnummer ein eigener Prozeß, aber das wäre bei vielen Startern nicht machbar. Daher werden die einzelnen Starter in einen Prozeß "gemultiplext". Über Details dieser Programmiertechnik mehr in Abschnitt 7.3.3.

Eine Empfangsschnittstelle existiert zum BEDIENER Prozeß, da z.B. bei Aufgabe oder Disqualifikation eines Teilnehmers dieses vom Bediener mitgeteilt werden muß. Von hier aus werden die Anzeigen für die einzelnen Sportler gesteuert, daher die Schnittstelle zu ANZEIGE. Meldungen von UEBERWACHUNG sind auch denkbar, z.B. um nach einem Fehlstart den betreffenden Teilnehmer zu sperren.

4. UEBERWACHUNG

Von hier werden alle das gesamte Rennen, d.h. die alle Teilnehmer betreffenden Abläufe gesteuert, etwa die Anzeige der zentralen Stoppuhr oder die Bewertung von Fehlstarts, oder der Abbruch bzw. das Ende der Veranstaltung.

5. ANZEIGE

Dieser Prozeß nimmt alle Aufträge zur Anzeige an der Stadiontafel entgegen und führt sie, unabhängig von der realisierten Anzeigen-Hardware, selbständig aus. Er nimmt auch Bewertungen vor, welche von mehreren gleichzeitigen Anforderungen priorisiert wird.

6. BEDIENER

Aufnahme und Verarbeitung von Bedienkommandos werden bearbeitet. Diese müssen zu jeder Zeit möglich sein und dürfen natürlich andere Abläufe wie Zeitmessung und -Anzeige nicht blockiern.

Durch die konsequente Anwendung des Prozeßkonzeptes ist es zu erreichen, daß beim Übergang von einer Sportart auf die andere lediglich der Prozeß UMWERT bzw. die Datenbasis geändert werden muß. Das Prinzip des Verbergens von Information ("information hiding") entkoppelt die Prozesse. Je komplexer Programmsysteme werden, desto wichtiger wird diese Fähigkeitdaß Änderungen im Programm auf einen engen, überschaubaren Bereich beschränkt bleiben.

Die Realisierung des Prozesses INDIV_RENNEN wird später (s. 7.2.2) als Beispiel für die Anwendung eines für das Prozeßkonzept entwickelten grafischen Hilfsmittel, nämlich SDL, dienen. Dort sind dann die sog. SDL-Diagramme oder Zustandsdiagramme dieses Prozesses zu finden.

6.2.2.2 Folgerungen aus dem Beispiel

Das Beispiel zeigt einige wichtige Eigenschaften eines modernen Software-Systementwurfes, die noch einmal zusammengefaßt werden.

1. Prozeßkonzept

Die Anwendung des Prozeßkonzeptes mit einem Realzeit-Betriebssystem erlaubt eine überschaubare Zergliederung der Aufgabe in einzelne Teilaufgaben. Dadurch bleibt nicht nur das Gesamtsystem übersichtlich, sondern es werden auch Test und Integration der einzelnen Teilaufgaben wesentlich erleichtert, indem z.B. für Testzwecke nur Teile des Systems zum Ablauf gebracht werden (vgl. auch 1.1.5).

2. Hierarchische Zergliederung der Aufgabe

Der Gesichtspunkt von verschiedenen Hierarchieebenen war eine wichtige Fragestellung bei der Aufteilung in einzelne Prozesse. Im Beispiel sind drei (Software-) Eingabe-Hierarchieebenen für die Verarbeitung von Sensoranreizen erkennbar (Abb. 6-4).

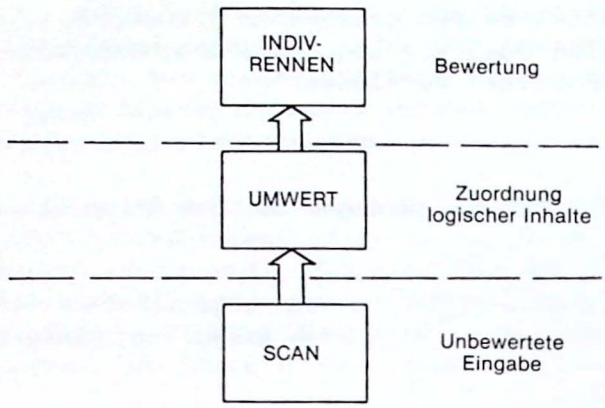

Abb. 6-4: Hierarchie bei der Bearbeitung
von Sensor-Eingaben

Weitere Unterteilungen sind denkbar, z.B. die Aufteilung von BEDIENER in eine unbewertete und bewertete Eingabe. Dasselbe gilt natürlich für die Ausgabe. Eng mit diesem Punkt ist ein weiterer Aspekt verknüpft:

3. Zeitliche Zergliederung der Aufgabe

Hiermit ist die zeitliche Entkopplung von einzelnen Teilaufgaben unter Realzeitgesichtspunkten zu verstehen, also die Aufteilung in zeitkritische und weniger zeitkritische Teile. Im Beispiel war die geforderte Zeitauflösung 1/1000 sec, so daß bei der Zahl von Sensoren für die zeitliche Erfassung eine Hardware-Lösung vorgeschlagen wurde. Als zeitkritischster Software-Teil wäre dann die Aufnahme und Weitergabe dieser von der Hardware-Logik abgelegten Sensor-Informationen, also der Prozeß SCAN.

Oberstes Gebot ist hier, daß keine Sensoränderungen verloren gehen dürfen. Auf diese Problematik werden wir noch eingehen (s. 6.2.3).

4. Verbergen von Information ("information hiding")

Die Schnittstellen der Prozesse untereinander werden so gewählt, daß so wenig wie nur irgend möglich über den inneren Aufbau der Prozesse nach außen dringt. Die Trennlinien der Prozesse sollen sich ausschließlich auf die logischen Funktionen beschränken, der Prozeß selbst erscheint als "black box". Diese vielleicht wichtigste Forderung im Software-Entwurf wird durch das Prozeßkonzept stark begünstigt. Man darf aber nicht dem Trugschluß unterliegen, daß dieses Ziel durch den Einsatz eines Realzeit-Betriebssystems automatisch erreicht wird. An der Realisierung der Schnittstellen zeigt sich, wie sorgfältig vor der eigentlichen Erstellung des Programms überlegt und abgesprochen wurde.

An dieser Stelle des Systementwurfes lohnt sich ein "Schnittstellen-Review", d.h. eine Begutachtung der abgesprochenen Schnittstellen auf das Prinzip des "information hiding" hin, und zwar durch Personen, die nicht die Entwickler der untersuchten Prozesse sind. Die Organisation dieses Reviews muß Teil des Software-Managements sein, was hier aber nicht Gegenstand der Betrachtung ist.

5. Wartbarkeit

Programmsysteme, die nach vorstehenden Regeln aufgebaut sind, vertragen durchaus drastische Änderungen, wobei die Änderungen auf überschaubare Bereiche beschränkt bleiben sollten. Dieser Punkt ist umso höher einzuschätzen, wenn man bedenkt, daß die Wartung eines Software-Paketes üblicherweise genausoviel kostet wie die Entwicklung selbst.

Nach diesem Beispiel, das die Methodik eines Software-Entwurfes zusammen mit einem Realzeit-Betriebssystem beleuchtet hat, werden wir auf weitere Teilaspekte des Software-Systementwurfes eingehen. Ein solch wichtiger Teil sind die Prioritäten, die den einzelnen Prozessen zugeordnet werden.

6.2.3 Prozeß-Prioritäten

Übliche Realzeit-Betriebssysteme sind prioritätsgesteuert, d.h. jedem Prozeß wird bei der Generierung im Process-Load-Table (s. 2.6.1) eine Priorität zugeordnet.

Die Verteilung von Prioritäten auf Prozesse wird oft intuitiv vorgenommen, geleitet von der gefühlsbetonten Vorstellung, der eine Prozeß habe "dringendere Aufgaben" zu erledigen als andere. Daß dieser Weg möglichst vermieden werden soll, da die Vergabe von Prioritäten eine kritische Angelegenheit ist, die auf das gesamte System in drastischer Weise wirken kann, soll gleich durch ein Beispiel belegt werden. Wir werden dann versuchen Hinweise zu geben, nach welchen Gesichtspunkten eine sinnvolle Prioritätenvergabe vorgenommen werden kann.

Für das folgende Beispiel sei folgender gebräuchlicher Scheduling-Algorithmus (s. 2.6.4.2) vorausgesetzt:

1. Ist ein Prozeß A im Zustand "running", und ein zweiter Prozeß B derselben Prioritätsstufe im Zustand "ready to run", so läuft A solange weiter bis er sich vertagt (z.B. im RECEIVE_EVENT, da seine Eingangswarteschlange leer ist). Erst dann läuft Prozeß B.

2. Ein Prozeß A kann durch einen höher-priorisierten Prozeß B, der "ready to run" ist, verdrängt werden, z.B. durch die Betriebssystem-Zeitunterbrechung oder indem A oder ein anderer Prozeß einen SEND_EVENT an B macht.

6.2.3.1 Bewertungskriterien

Betrachten wir ein (vereinfachtes) Beispiel aus der Kommunikationstechnik, nämlich eine rechnergesteuerte Telefonvermittlung. Die Software dieses Systems besteht aus folgenden Komplexen (Abb. 6-5):

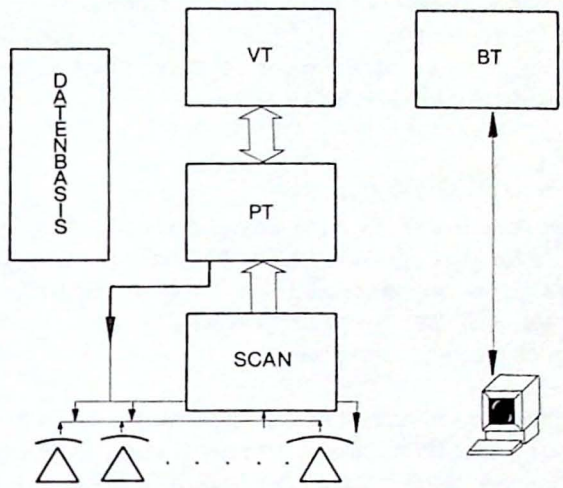

Abb. 6-5: Software-Komplexe einer Telefonvermittlung (vereinfacht)

Die ⏃ Symbole stellen die Telefonapparate, d.h. die Fernsprechteilnehmer dar. SCAN hat die übliche Funktion: er fragt einen Apparat nach dem anderen ab, ob Änderungen signalisiert werden, z.B. Hörer abheben oder Ziffern wählen. PT (=Peripherie-Technik) bekommt die von SCAN erkannten Anreize und meldet sie im normalen Telefonbetrieb an VT (=Vermittlungstechnik) weiter, und zwar in geräteunabhängiger Form. Das heißt, daß PT an VT immer meldet "Hörer abgehoben", egal ob durch verschiedene Fernsprechertypen das Abheben durch verschiedene Codes signalisiert wird. Insofern entspricht PT dem Prozeß UMWERT aus dem Beispiel in Abschnitt 6.2.2.1. PT macht auch Einstellungen an die Geräte, z.B. der VT Befehl "Klingeln" wird von PT in die gerätespezifische Form umgesetzt und an das Endgerät weitergegeben. VT gibt an PT z.B. auch die Aufträge, die Sprechwege zweier Teilnehmer zusammenzuschalten.

Der Prozeß VT ist PT übergeordnet und wickelt die Logik des Fernsprechbetriebes ab.

Der Prozeß BT (=Betriebstechnik) behandelt alle Ein-/Ausgaben vom und zum Bedienblattschreiber, wie Einrichten neuer Teilnehmer oder Änderung der Rufnummer eines Teilnehmers.
Wie sollen die Prioritäten der Prozesse vergeben werden? Zwei Wege werden gezeigt, und zwar zuerst die "intuitive" Lösung und danach die "pragmatische" Lösung.

1. Intuitiver Weg

Unbefangen würde man urteilen, der Prozeß SCAN muß die höchste Priorität haben, da auf jeden Fall Aktionen der Teilnehmer möglichst schnell erkannt werden sollen. Als nächste Prioritätsstufen werden PT, VT und BT vorgeschlagen, da in dieser Reihenfolge die Realzeitanforderungen abnehmen.

Diese Lösung wird sicher arbeiten, solange keine besondere Lastsituation vorliegt. Es wäre aber vermutlich derselbe Effekt, wenn die Prioritäten genau umgekehrt vergeben wären, da ohne Belastung das Zeitverhalten des Programmsystems unkritisch ist. Schwieriger werden die Verhältnisse, wenn eine Lastsituation vorliegt, d.h. wenn viele Teilnehmer gleichzeitig telefonieren wollen. Beim vorgeschlagenen Prioritätsschema SCAN-PT-VT-BT führt das dazu, daß alle Aktionen der Teilnehmer von SCAN aufgenommen werden; im Extremfall gibt SCAN den Prozessor nicht mehr ab. Das Ergebnis ist ein Meldungsstau in der Eingangswarteschlange von PT, in die SCAN ja die Teilnehmeranreize einträgt. Das kann dazu führen, daß das System "abstürzt", da von SCAN ständig neue Poolstücke geholt werden, aber nie welche freigegeben werden, da der Zielprozeß, nämlich PT, nie zum Zuge kommt.

Dasselbe gilt im Verhältnis PT-VT. In einer Lastpause, in der keine neuen Anreize anfallen, arbeitet PT seine Eingangswarteschlange ab und füllt die von VT, wo die eigentlichen Vermittlungsfunktionen stattfinden. Hält die Lastsituation dagegen an, ist der VT-Prozeß praktisch tot, da er geringe Priorität hat. Das heißt aber, daß keine Gespräche mehr vermittelt werden.

Man könnte nun ebenso argumentieren und sagen, soll die Prioritätsreihenfolge doch VT-PT-SCAN sein, in absteigender Reihenfolge. Dieses Schema hätte den Vorteil, daß wenn ein Anreiz aufgenommen wird, er auch sicher bearbeitet wird.

Hier kommt dann allerdings der Realzeitgesichtspunkt zum Tragen. Falls PT oder VT längere Programmstücke durchlaufen, etwa Suchprogramme, kann es vorkommen, daß Teilnehmeranreize verloren gehen, da SCAN eventuell zwischen zwei Aktionen des Teilnehmers nicht an die Reihe kommt. So können Wahlziffern verloren gehen und Falschwählungen passieren. Das ist natürlich ein völlig unhaltbarer Zustand und bei weitem ärgerlicher, als wenn der Teilnehmer Wartezeiten in Kauf nehmen muß.

Ebenfalls problematisch ist die Priorität des BT-Prozesses. Bekommt er niedere Priorität, wird er im Lastfall nie bedient, d.h. es sind keine Eingaben und Ausführungen von Bedienkommandos mehr möglich, oder nur noch sehr schleppend. Bekommt er höhere Priorität, müssen die anderen Prozesse warten, wenn Bedienkommandos abgearbeitet werden.

Wir würden folgende Lösung des Problems vorschlagen:

2. Pragmatischer Weg

SCAN muß die höchste Priorität behalten, damit keine Anreize verlorengehen. Anreizschwemmen im Überlastfall muß man im angeführten Fall anders begegnen: die Signalisierungen der Teilnehmer, die im Gesprächsaufbau (z.B. Wahl) oder im Gespräch sind, werden vom System akzeptiert. Teilnehmer, die neu ein Gespräch aufbauen wollen (z.B. durch Hörer abheben) werden für eine kurze Zeit (z.B. 5-10 sec) so gesperrt, daß SCAN keine Signalisierung mehr bekommen kann. Beim Systementwurf sollte man darauf achten, daß solche Sperren möglich sind. Dieses Verfahren ist ein Beispiel einer Lastabwehr. PT sollte die niederste Priorität bekommen, VT eine Zwischenpriorität.

Der Ablauf sieht dann so aus:

SCAN meldet alle Anreize der Teilnehmer (von Lastabwehr einmal abgesehen). Falls VT noch aktiv war, wird diese Aktivität beendet. Erst dann kann PT zum Zug kommen, was wieder als Ausgangspunkt für neue VT-Aktionen dient.

Dieses Prioritätsschema hat folgende Vorteile:

1. Alle Anreize werden erkannt.

2. Kein stoßweiser Betrieb zwischen PT und VT, sondern gleichmäßige Abarbeitung von Anreizen.

Die Priorität des Prozesses BT sollte gleich gewählt werden wie PT, damit auf jeden Fall betriebstechnische Ein- und Ausgaben noch möglich sind, wenn auch dadurch die Wartezeit der Teilnehmer größer wird. Folgende Tabelle faßt alles noch einmal zusammen. Die Zahlen geben die gewählten Prozeß-Prioritäten an, hohe Zahlen entsprechen hohen Prioritäten:

Tabelle 6-1

SCAN	PT	VT	BT	Kommentar
4	3	2	1	"intuitive" Lösung, Meldungsstau bei Last.
2	3	3	1	Anreizverluste unter Last möglich, kein Meldungsstau.
4	2	3	1	kein Meldungsstau zwischen PT und VT, keine Anreizverluste, BT-Aufträge unter Last nicht möglich.
3	1	2	1	BT-Aufträge unter Last möglich. Bestmöglicher Kompromiß.

Hier soll nochmals betont werden, daß das gewählte Prioritätsschema stark vom Scheduling-Algorithmus abhängt. Der Scheduling Algorithmus war: Höher-priorisierte Prozesse verdrängen niedriger-priorisierte, bei gleichen Prioritäten läuft ein Prozeß solange, bis seine Eingangswarteschlange abgearbeitet ist.

Ein anderer Scheduling-Algorithmus wäre z.B. "round-robin": bei gleich-priorisierten Prozessen wird ein Element der Eingangswarteschlange abgearbeitet, danach wird auf den nächsten Prozeß umgeschaltet. Bei "round robin" Scheduling könnte in unserem Beispiel PT und VT dieselbe Priorität bekommen, da auch so der schubweise Betrieb unterbunden wäre.

Dies ist ein Beispiel, bei dem der Entwickler von Anwendungsprozessen (z.B. VT) über Betriebssystemdetails Bescheid wissen muß, um das Verhalten des Systems interpretieren zu können.

6.2.3.2 Änderung von Prioritäten im Betrieb

Im vorigen Beispiel hatten wir den Prozeß BT, der Bediener Ein-/Ausgaben abwickelt. Diesem Prozeß wurdegn dieselbe Priorität wie PT gegeben, damit im Lastfall Ein-/Ausgaben noch möglich sind. In diesem, sowie in anderen Fällen, wo höher-priorisierte Prozesse an niedriger-priorisierte Nachrichten senden, die dann der niederen Priorität wegen nicht abgearbeitet werden können, wäre auch ein anderes Verfahren denkbar. Man könnte zeitweise die Priorität dieser Prozesse anheben, um sie auch zum Zug kommen zu lassen. Als Grund für ein solches Vorgehen kann zweierlei angeführt werden:

1. Auch unter Last sollen niedrig-priorisierte Prozesse ihre Aufgaben erledigen können.

2. Ein einseitiger Meldungsfluß von einem hoch- zu einem niedrig-priorisierten Prozeß kann dazu führen, daß irgendwann keine Poolelemente für Meldungen zur Verfügung stehen, das System ist dann nicht mehr lebensfähig.

Eine solche dynamische Zuteilung von Prioritäten setzt in unserem Beispiel allerdings voraus, daß das Betriebssystem periodisch die "Pegelstände der Prozeß-Eingangswarteschlangen abprüft, bei "Hochpegel" die Priorität anhebt und bei "Niedrigpegel" auf den alten Wert absenkt. Dabei darf die Priorität nicht prinzipiell auf den höchsten Wert angehoben werden, sondern nur so hoch, daß das Realzeitverhalten nicht gefährdet ist.

Durch den notwendigen Verwaltungsaufwand wird das System zusätzlich belastet, so daß versucht werden sollte, das Problem durch entsprechende Zuteilung statischer Prioritäten zu lösen.

Die Bevorzugung einzelner Prozesse, durch Vergabe hoher Prioritäten, kann auch noch zu einem anderen Zweck entfremdet werden, nämlich zur Lösung von Konflikten, die beim gleichzeitigen Zugriff mehrerer Prozesse auf gemeinsame Betriebsmittel auftreten,

z.B. auf gemeinsame Datenbestände, die sog. "Datenbasis" (s. 6.3). Die übliche Methode zur Lösung dieser Konfliktfälle ist das Prinzip der "kritischen Region", wobei der Anwender mittels spezieller Betriebssystemaufrufe (s. 2.5.3) das Zugriffsrecht vom Betriebssystem erteilt bekommt.

Man kann das Problem auch einfacher lösen, nämlich indem der Prozeß vor dem Zugriff seine Priorität heraufsetzt, sich dadurch die Exklusivität sichert, und sie nach dem Zugriff wieder zurücksetzt. Damit das funktioniert, muß folgendes erfüllt sein:

1. Die Laufzeit eines Prozesses unter heraufgesetzter Priorität darf nur so kurz sein, daß das Systemverhalten als Ganzes nicht gestört wird. Im vorigen Beispiel etwa darf SCAN nicht solange verdrängt werden, daß Anreize verloren gehen können.

2. Das Betriebssystem muß dem Anwender eine geeignete Schnittstelle zur Verfügung stellen, z.B. in Form der Aufrufe SET_PRIORITY und RESET_PRIORITY (S. 3.1.4.2).

Der Vorteil der Methode liegt darin, daß kein aufwendiger Betriebssystem-Überbau zur Verwaltung kritischer Regionen erforderlich ist.

6.2.3.3 Zeitliche Verschachtelung der Prozesse

Das Zusammenspiel von Prozeß-Prioritäten und Scheduling-Algorithmus setzt ein Umdenken des Programmierers voraus, um seinen Programmteil in den gesamten zeitlichen Ablauf des Systems einordnen zu können. Unterschiedliche Prozeßprioritäten bewirken eine zeitliche Verwürfelung in der Abarbeitung von Aufträgen. Ein Beispiel:

Zwei Prozesse A und B greifen beide auf gemeinsame Daten zu. A durchläuft ein Programmstück:

```
                    :
                    :
              Datenbasiszugriff
                    :
                    :
          SEND_EVENT (an Prozeß B)
                    :
                    :
    Aktion aufgrund obigen Datenbasiszugriffes
                    :
                    :
```

Falls B höhere Priorität hat, als A, findet an der Stelle des SEND_EVENT ein Prozeßwechsel statt, es läuft Prozeß B. Falls B die Datenbasis manipuliert, führt A wenn er wieder an der Reihe ist, möglicherweise falsche Aktionen aus, da zwischen A's Datenbasiszugriff und der resultierenden Aktion ein Prozeßwechsel lag. Durch die zeitliche Verzahnung der Prozesse aufgrund unterschiedlicher Prioritäten können also Fehler entstehen. Dieses Beispiel darf übrigens nicht mit dem vorigen verwechselt werden, wo zur Vermeidung von Konfliktfällen beim Datenbasiszugriff die Prioritäten kurzzeitig angehoben wurden. Das ist nur während des Datenbasiszugriffes der Fall. Beim SEND_EVENT hat A längst wieder seine alte Priorität.

6.2.3.4 Zusammenfassung "Prioritäten"

1. Keine "intuitive" Vergabe von Prioritäten, sondern mögliche Konflikt- und Extremfälle im Gedankenexperiment durchspielen und als Kriterium für die Prioritätenverteilung heranziehen.

2. Prioritätenvergabe im engen Zusammenhang mit dem Scheduling-Algorithmus vornehmen.

3. Änderungen von Prioritäten im Betrieb nur in Ausnahmefällen vornehmen und wenn sichergestellt ist, daß das Systemverhalten nicht negativ beeinflußt wird.

4. Die Gefahr der zeitlichen Verwürfelung verschiedener Prozesse durch unterschiedliche Prioritäten bedenken.

Diese Punkte müssen in der Entwurfsphase durchgespielt werden. Ist das System erst programmiert, sind Änderungen, z.B. von Prozeßscnittstellen, oft nicht mehr möglich.

6.2.4 Realzeit-Aspekte

6.2.4.1 Interrupts

Realzeit-Systeme sind unterbrechungsgesteuert. Die Interrupt-Programme, die entweder aufgrund eines einmaligen Ereignisses spontan ablaufen können oder periodisch von einem Zeitgeber angestoßen werden, bilden das Bindeglied zur tatsächlich verstreichenden Zeit.

Interrupts können im Lauf eines Prozesses zu beliebigen Zeiten eintreffen, wenn der Prozeß nicht unter Interrupt-Sperre läuft. Um diese Zeit der Unterbrechung kurz zu halten, wendet man üblicherweise folgendes Verfahren an (Abb. 6-6):

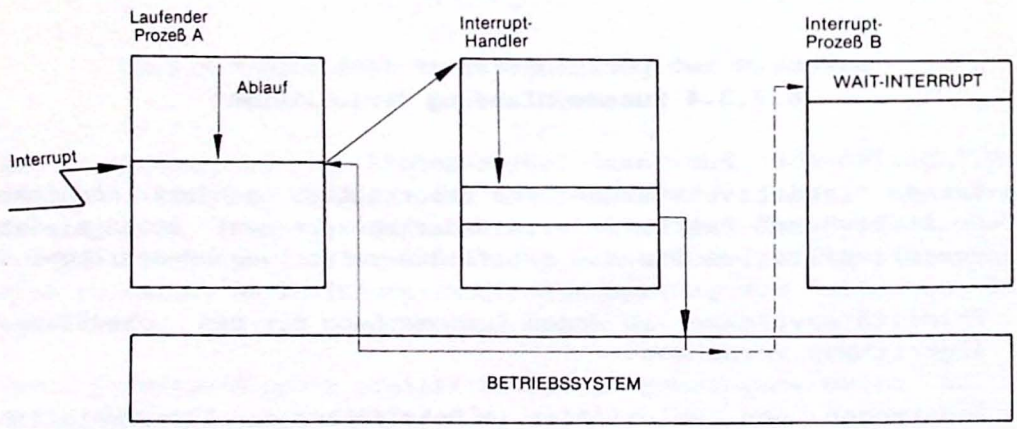

Abb. 6-6: Zusammenwirken von Interrupt-Handler und Interrupt-Prozeß

Man teilt die gesamte Abarbeitung der Unterbrechung in zwei Teile. Der eine Teil, das Interrupt-Programm, oft auch "Interrupt-Handler" genannt, ist der Teil, der unmittelbar nach der Unterbrechung angesprungen wird. Dieser ist zeitoptimiert und nimmt nur die nötigsten Aktionen zur Sicherstellung der Daten vor, die beim Interrupt anfallen. Die logische Bearbeitung der Daten geschieht dann im zweiten Teil, dem Interrupt-Prozeß, der vom Interrupt-Handler benachrichtigt wird. Dieser Prozeß wird dann vom Betriebssystem entsprechend seiner Priorität aktiviert und setzt die Interruptbearbeitung auf logischer Ebene fort. Zur Verständigung zwischen Interrupt-Handler und -Prozeß dienen die Betriebssystem-Aufrufe SIGNAL_INTERRUPT und WAIT_INTERRUPT (s. 3.1.3.3). Mit SIGNAL_INTERRUPT benachrichtigt der Interrupt-Handler den Prozeß, der sich entsprechend mit WAIT_INTERRUPT vertagt hatte.

Betriebssystem-Aufrufe laufen meistens unter Interrupt-Sperre. Beispielsweise werden während des Ein- und Auskettens von Poolstücken in Warteschlangen, die Verknüpfungszeiger in Poolstücke geändert. In dieser kritschen Zeit darf niemand versuchen (etwa durch eine weitere, von einer Unterbrechung herrührende Aktion) die Zeiger zu ändern, die erst teilweise aktualisiert sind.

Werden von Anwenderprozessen Interrupts gesperrt, ist streng darauf zu achten, daß nicht durch zu langes Sperren Fehlreaktionen des Systems passieren. Als Beispiel sei an den Telefoncomputer der Einführung erinnert. Dort sollten die zu wählenden Ziffern durch Öffnen und Schließen eines Wahlkontaktes dem Vermittlungsamt signalisiert werden. Kommen die diesen Kontakt steuernden Zeitgeber-Interrupts wegen zu langer Interrupt-Sperre, z.B. 100 msec nicht durch, können Falschwählungen passieren.

Eine interessante Systemeigenschaft ist die minimale Zeit, in der das System auf Unterbrechungen reagieren kann. Hier sind verschiedene Parameter zu berücksichtigen:

1. Die längste Zeit, in der Unterbrechungen gesperrt sein können.

2. Die Priorität der Unterbrechung. Sie ist hardware-gesteuert und nicht zu verwechseln mit der Prozeßpriorität.

3. Die Zeit, die das Interrupt-Programm braucht, um den aktuellen Prozessorstatus zu speichern.

4. Die Zeit vom Einsprung in das Interrupt-Programm bis das Anwenderprogramm die Aktion ausgeführt hat.

Falls für Punkt 1 die Betriebssystem-Aufrufe maßgebend sind (z.B. SEND_EVENT mit Prozeßwechsel) ist eine typische Zeit, je nach Betriebssystem, Prozessorgeschwindigkeit und Wartezyklen 500 usec - 1 msec (z.B. im Betriebssystem iAPX86/30 braucht der SEND_MESSAGE Aufruf ca. 500 usec bei 8 MHz 8086 ohne Wartezyklen).

Für Punkt 2 nehmen wir der Einfachheit halber höchste Priorität an, so daß hier kein Zeitverzug ensteht. Für niedrig-priorisierte Interrupts muß hier die längste Laufzeit der höher-priorisierte Interrupts, einschließlich eventueller Schachtelungen, genommen werden.

Für Punkt 3 liegt die Zeit typisch bei 50-100 usec.

Die Zeit für Punkt 4 hängt vom einzelnen Programm ab. Zeitkritische Interrupt-Programme sollten in Assembler-Code geschrieben werden, der auf günstige Laufzeit optimiert wurde. Nehmen wir als Beispiel ein kleines Programm mit 50 Assemblerbefehlen, mit grob geschätzten 5 usec pro Befehl, haben wir 250 usec.

Addieren wir alle Zeiten, kommen wir auf eine Größe zwischen 800 usec und 1.3 msec. Das bedeutet, daß kürzere Systemreaktionen im "worst-case" auch für Unterbrechungen höchster Priorität nicht zu erreichen sind. Einen großen Anteil an dieser Zeit hat das Betriebssystem. Hier sind durch "maßgeschneiderte" Betriebssysteme, bei denen z.B. die Aufrufe nicht prinzipiell unterbrechungsgeschützt sind, sondern nur die wirklich kritischen Teile, deutliche Verbesserungen zu erreichen.

6.2.4.2 Einfluß des Betriebssystems auf die Systemdynamik

Der Komfort, den ein Betriebssystem bietet, ist natürlich nicht unnötig. Wie wir gesehen haben, ist er ein bestimmender Faktor des Realzeitverhaltens. Aber auch die allgemeine "Dynamik" des Systems, d.h. die Geschwindigkeit mit der Anwenderprogramme ab-

laufen, wird vom Betriebssystem beeinträchtigt. Als Richtzahl kann gelten, daß ca. 20-30% der gesamten Prozessorzeit vom Betriebssystem beansprucht werden. Daher ist es sinnvoll, wenige Betriebssystem-Aufrufe zu benutzen. Das aber bedeutet möglichst schmale Schnittstellen zwischen den Prozessen. Außerdem sollte die Zahl der Prozesse auf das notwendige Minimum reduziert bleiben.

6.2.5 Prozeß-Überwachungen

Eine der wichtigsten Forderungen an ein automatisch laufendes System ist, daß es unter allen möglichen Umständen zuverlässig arbeitet. Einige Mittel, den ordungsgemäßen Ablauf der Software zu überwachen, sind schon angesprochen worden, z.B. der "Watchdog" (s. 5.2.3) oder die Betriebssystemüberwachungen (s. 2.9). An dieser Stelle soll beispielhaft auf zwei weitere Möglichkeiten hingewiesen werden, die innerhalb von Anwenderprozessen eingesetzt werden können.

6.2.5.1 Zeitüberwachungen

Das Prinzip ist einfach: Erwartet ein Prozeß auf eine Mitteilung an einen anderen Prozeß eine Antwort, geht er davon aus, daß diese nach Ablauf einer Maximalzeit eingetroffen sein muß. Er gibt also zu Beginn der Wartezeit einen Zeitauftrag an das Betriebssystem (TIMER_ASSIGN). Trifft die Reaktion des Partners vor Ablauf der Zeit ein, wird der Zeitauftrag gelöscht und alles ist in Ordnung. Wartet er jedoch vergebens, läuft sein Zeitauftrag ab, und er bekommt vom Betriebssystem Nachricht, daß die angeforderte Zeit um ist. Daraufhin kann der Prozeß in eine Fehlerbehandlung eintreten, etwa einem Überwachungsprozeß Meldung machen, eine LED einschalten, einen Ausdruck veranlassen oder sonst notwendige Aktionen vornehmen.

Auf jeden Fall ist erreicht, daß der Prozeß nicht zeitlich unbegrenzt wartet und so für andere Aufgaben blockiert wäre. Bei korrektem Ablauf der Prozesse würde natürlich jede erwartete Quittung auf eine Meldung auch ankommen, sofern nicht externe

Hardware oder Software beteiligt ist. Bei einem Multiprozessorsystem könnte ein Prozessor wohl noch intakt, ein zweiter, von dem eine Quittung erwartet wird, dagegen ausgefallen sein. Daher wird diese Überwachungsmöglichkeit innerhlab der Prozesse eines Prozessors nicht empfohlen. Die vielen Timer, die nie ablaufen, bedeuten einen unnötigen Aufwand, der das Programmsystem verlangsamt. Bei Quittungen von externen Prozessoren oder Reaktionen von Geräten ist die Methode dagegen sehr wirkungsvoll, um Blockierungen von Prozessen zu verhindern.

6.2.5.2 Plausibilitätskontrollen

Prozesse sollten mit den Meldungen, die sie gesendet bekommen, Plausibilitätskontrollen anstellen. Meistens werden bestimmte Meldungen von anderen Prozessen, abhängig vom Prozeßzustand, (s. 7.1.1) erwartet. Trifft eine Meldung ein, die eigentlich nur in einem anderen Zustand kommen dürfte, kann wieder ähnlich wie im vorigen Beispiel verfahren werden. Auch dieser Fall dürfte bei ausgetestetem Programmsystem eigentlich nicht vorkommen, außer bei Beteiligung externer Prozesse. Diese Überprüfung der Meldungen bedeuten jedoch im Gegensatz zum vorigen Beispiel keine Verlangsamung des Systems, sondern nur einen zusätzlichen Programmzweig, der im Normalfall nie durchlaufen wird.

Damit ist der Teil abgeschlossen, der Probleme des Entwurfes und der Implementierung von Prozessen behandelt. Wir wenden uns nun einem speziellen Teil der Software zu, nämlich der Datenbasis.

6.3 Die Datenbasis

Unter einer Datenbasis versteht man alle Daten eines Systems, die nicht lokal einem Prozess zugeordnet werden können, sondern globale, d.h. systemweite Bedeutung haben. In der Praxis heißt das, daß mehrere Prozesse auf die Datenbasis zugreifen, sowohl lesend als auch schreibend.

6.3.1 Zugriff auf gemeinsame Daten

Es ist gefährlich, den Anwenderprozessen direkten Zugriff zu den Daten einer Datenbasis zu gewähren, da verschiedene Konfliktfälle zu Fehlern führen können, z.B:

- Es können versehentlich durch Programmierfehler Daten der Datenbasis fehlerhaft von einem Prozeß geändert werden.

- Es können beim gewollten Ändern von Datenbeständen durch einen Prozeß Konfliktfälle auftreten, wenn ein anderer Prozeß gleichzeitig liest und dadurch auf erst teilweise geänderte Daten zugreift, was zu Inkonsistenzen und Fehlern führt.

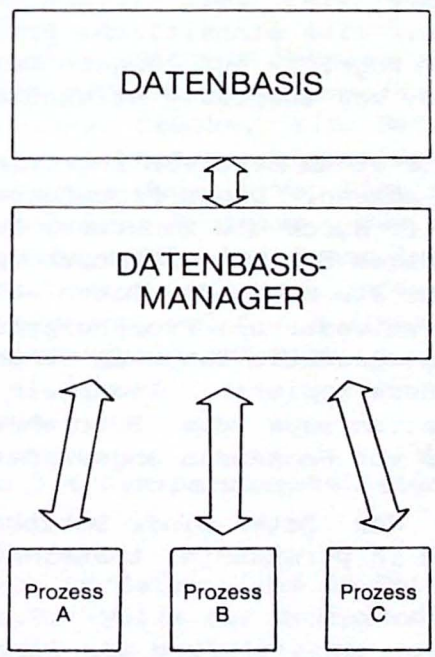

Abb. 6-7: Datenbasis-Manager regelt gleichzeitigen Zugriff der Prozesse auf die Datenbasis

Um den direkten Zugang der Anwender zu den Daten zu verhindern, empfiehlt sich der Einsatz eines Datenbasisverwalters oder "Data Base Managers (DBM). Es handelt sich um ein Programm, das dem Anwender den Zugang zu den Daten ermöglicht und garantiert, daß keine Inkonsistenzen auftreten, gleichzeitig aber die Daten vom direkten Zugriff durch den Anwender abschirmt (s. Abb. 6-7).

Ein DBM ist auf verschiedene Weise zu realisieren. Hier zwei Beispiele:

1. Der DBM kann ein eigener Prozeß sein, der mit den Benutzerprozessen über Meldungen verkehrt, in denen die gewünschten Daten ausgetauscht werden. Das Problem des wechselseitigen Ausschlußes des Zugriffs auf die Daten ist optimal gelöst, da der DBM nur eine Anforderung zu einer Zeit bearbeitet. Bei häufigen Zugriffen auf die Datenbasis ist diese Methode jedoch schwerfällig, da durch den Betriebssystem-Überbau des SEND-RECEIVE Mechanismus viel Zeit verbraucht wird. Außerdem müssten, falls nur eine einheitliche Poolstückgröße zur Verfügung steht, für Zugriffe auf längere Datenfelder die Daten zerstückelt werden, was ebenfalls Aufwand bedeutet.

2. Der DBM besteht aus einem Satz von Zugriffsprozeduren, die die Benutzer anwenden müssen. Diese Prozeduren schützen die Daten vor direktem Zugriff durch die Benutzer. Den Benutzerprozessen müssen nur die Namen der Zugriffsprozeduren bekannt gemacht werden, und nicht die Namen der Daten selbst. Der Anwender bekommt die Daten entweder als Prozedurparameter zurück, oder der DBM kann die gewünschten Daten in einen lokalen Speicherbereich des Anwenders kopieren. Innerhalb der Prozeduren kann der Semaphore-Mechanismus des Betriebssystems zum gegenseitigen Ausschluß von Prozessen angewendet werden.

Zur Klassifizierung der Daten einer Datenbasis kann man eine Dreiteilung vornehmen in permanente, transiente und semipermanente Daten.

6.3.2 Permanente Daten

Es handelt sich um Daten, die während des Betriebes nur gelesen werden können. Sie werden bei der Systemgenerierung in den Speicher z.B. EPROM oder einen schreibgeschützten Teil des System Memory eingebracht. Diese Daten sind immer konsistent, da sie im Betrieb nie geändert werden, es müssen daher auch keine Maßnahmen zum gegenseitigen Ausschluß von Prozessen bei gleichzeitigen Zugriffsversuchen ergriffen werden. Ein DBM ist hier nötig, wenn nicht alle Daten allen Anwendern zugänglich sein sollen. Der DBM hat die Aufgabe, die Zugriffsberechtigung zu prüfen, die Berechtigung zu erteilen oder zu verweigern.

6.3.3 Transiente Daten

Diese Daten werden sehr häufig im Betrieb gelesen und geändert. Hier müssen nur selten größere Datenbereiche gemeinsam behandelt werden, sondern nur kurze Stücke, z.B. Betriebszustände von Geräten, auf die mehrere Prozesse zugreifen. Die Zugriffe sind daher meist nur von kurzer Dauer. Man kann also bei gleichzeitigem Zugriffsversuch mehrerer Prozesse, den einen warten lassen. Mehrere gleichzeitige Zugriffe, bei denen lediglich gelesen wird, braucht man natürlich nicht auszuschließen. Für jeden unterschiedlichen Zugriffswunsch, also z.B. lesen oder ändern, sollte eine eigene Zugriffsprozedur existieren.

6.3.4 Semipermanente Daten

Hier handelt es sich um Daten, die im Betrieb selten geändert werden. Es können z.B. Teile der Systemkonfiguration sein, die nur vom Bedienterminal aus modifizierbar sein soll. Man denke etwa an die Rufnummern von Fernsprechteilnehmern, bei einer rechnergesteuerten Fernsprechvermittlung. Diese Daten gehören oft zu längeren zusammenhängenden Datenfeldern. Im erwähnten Beispiel gehört die Rufnummer eines einzelnen Teilnehmers zum gesamten Rufnummernspeicher der Anlage. Wird jetzt eine Rufnummer geändert, muß der gesamte Rufnummernspeicher für Leseversuche ge-

sperrt werden, bis die Änderung abgeschlossen ist. Sind mit der Änderung etwa noch zusätzliche Sortiervorgänge verbunden, kann die Zeit der Sperre unakzeptabel lang werden.

Hier kann ein einfaches Verfahren angewendet werden, um lesende Prozesse während des Schreibens anderer Prozesse nicht zu blockieren (Abb. 6-8):

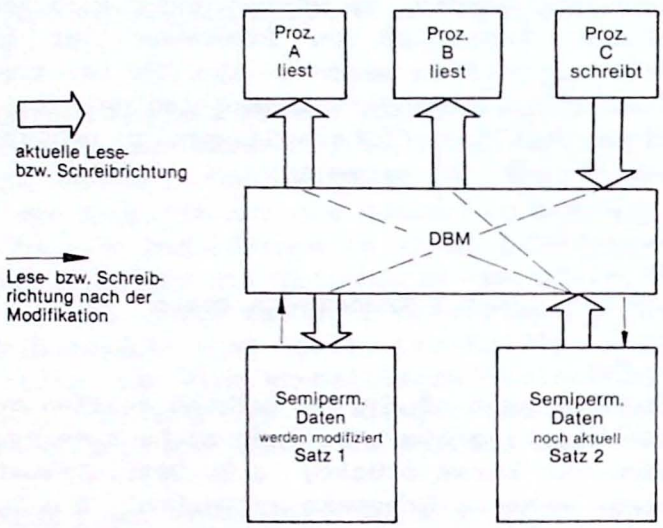

Abb. 6-8: Kurze Totzeiten der semipermanenten Daten durch zwei Datensätze

Es werden zwei identische Abbilder der semipermanenten Daten im Speicher abgelegt, wobei auf eines lesend zugegriffen wird. Im anderen Abbild kann eine Änderung vorgenommen werden, die jetzt ruhig auch länger dauern darf. Nach der Änderung wird abgewartet, bis auf das erste Abbild kein Anwender zugreift, z.B. indem momentan laufende Zugriffe noch beendet werden, aber keine neuen Anforderungen mehr akzeptiert werden. Danach werden "Lese"-Abbild und "Schreib"-Abbild vertauscht, indem fortan bei Lesezugriffen auf das vorherige und jetzt sich auf dem neuesten Stand befindliche "Schreib"-Abbild zugegriffen wird. Das jetzige und veraltete "Schreib"-Abbild muß danach natürlich ebenfalls aktualisiert werden. Der Vorteil ist, daß die Totzeit, in der keiner lesend auf die semipermanenten Daten zugreifen darf, klein gehalten wird. Der Nachteil ist, daß durch die zwei Abbilder ein erhöhter Speicherplatzbedarf besteht.

Auf weitere Eigenschaften von Datenbasen, z.B. Ordnungsprinzipien von Datenbanken soll hier nicht eingegangen werden. Stattdessen werden wir uns einigen Software-Aspekten des Multiprozessing zuwenden.

6.4 Multiprocessing

Die grundlegenden, hardwarenahen Probleme des Zugriffs mehrerer Rechner auf gemeinsame Betriebsmittel, wie z.B. Speicher, wurde in Abschnitt 5.4 beschrieben. Hier soll nun auf einige Software-Aspekte von Multiprocessor-Systemen oder sog. "verteilten Systemen" eingegangen werden. Systeme in "loser Kopplung (s. 5.4.3) verkehren effektiv mittels Meldungsaustausch über das Kommunikationssystem (s. 1.1.4), welches das Datenübertragungsprotokoll abwickelt. An dieser Stelle soll jedoch auf die Problematik von Multiprozessorsystemen in "fester Kopplung" (s. 5.4.2) eingegangen werden, also solchen Systemen, bei denen verschiedene Prozessoren über einen gemeinsamen Systembus direkt auf einen gemeinsamen Speicher, das System-Memory zugreifen.

Einige der angesprochenen Punkte in diesem Kapitel gehören logisch zum Betriebssystem-Kern. Um das Thema "Multiprocessing" nicht zu zerstückeln, wird es hier als Ganzes behandelt.

6.4.1 Funktionsteilung ("Function Sharing")

Bei einem Multiprozessorsystem dieser Art, ist an einen Prozessor die Ausführung einer bestimmten Funktion gebunden. Es ist denkbar, daß Ein-/Ausgabe Prozessoren vorhanden sind, oder daß ein Datenbasis-Manager einen eigenen Prozessor erhält. Allgemein könnte man jedem Prozeß in einem Multitasking-Betriebssystem einen eigenen Prozessor zuordnen. Dann wäre für jede Prozeß-Eingangswarteschlange ein Prozessor zuständig. Das Programm einschließlich dem Betriebssystem und allen Daten, die nicht von allgemeinem Interesse sind, können im lokalen Speicher der einzelnen Prozessoren liegen.

Im System Memeory müssen insbesondere die Prozeß-Eingangswarteschlangen liegen, da ja verschiedene Prozessoren Meldungen austauschen.

Der Vorteil ist die relativ geringe Belastung des System-Bus. Das Verfahren ist verbreitet zur Realisierung von Ein-/Ausgabefunktionen wie Datenübertragung oder Dateibehandlung, die dann meist auch auf spezieller Hardware ablaufen.

Verteilt man einzelne Prozesse eines Programmsystems auf unterschiedliche Prozessoren, ist zumindest ein Teil des Systems außer Betrieb, wenn ein Prozessor ausfällt. Wünschenswert wäre es, wenn sich mehrere Prozessoren, die in Hard- und Software identisch sind, die anfallenden Aufgaben teilten. Bei diesem Verfahren spricht man von Lastteilung.

6.4.2 Lastteilung ("Load Sharing")

In Abschnitt 5.4.4 wurde das EWSD-Fernsprechvermittlungssystem als Beispiel für eine komplexe Anwendung von Mikrocomputer-Steuerungen aufgeführt. Dort wurde das Beispiel "Lastteilung" anhand der einzelnen Line Trunk Groups (LTGs) angeführt, von denen jede in Hard- und Software identisch ist, und für eine bestimmte Zahl von Teilnehmern zuständig ist. Diese statische Aufteilung der Last in einem lose gekoppelten System unterscheidet sich von der dynamischen Aufteilung der Last im Betrieb eines fest gekoppelten Systems, worauf wir jetzt eingehen werden.

Zur Erinnerung sei nochmals auf Abb. 5-9 verwiesen, welche die Hardware-Struktur des Multiprozessorsystems in "fester Kopplung" darstellt. Jeder Prozessor soll jede Aufgabe erfüllen können. Deshalb darf keiner gegenüber dem anderen Vorrang haben. Da jeder alles können soll, müssen alle Daten jedes Prozessors im gemeinsamen System Memory liegen. Daher dürfen diese Daten, auf die ja mehrere zugreifen, nur über Verriegelungsmechanismen gelesen bzw. geschrieben werden, um Inkonsistenzen zu vermeiden.

Abb. 6-9 illustriert den Vorgang der Lastteilung. Jeder Rechner R1...Rn läuft unabhängig vom anderen, hat also insbesondere einen eigenen Taktgenerator. Die Hardware-Anreize sollen von einem Interface-Prozessor IOP in ein Dual Port RAM eingetragen werden.

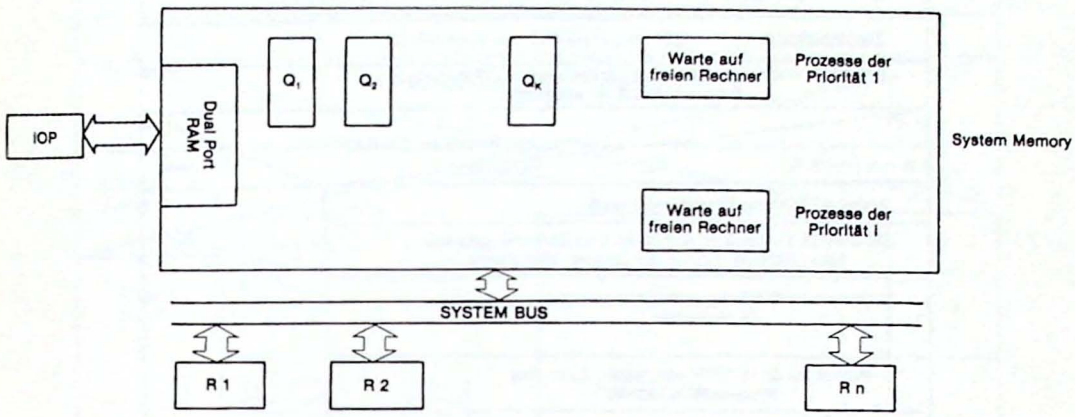

Abb. 6-9: Multiprozessorsystem in Lastteilung

Q1...Qk bezeichnen die Eingangswarteschlangen der einzelnen Prozesse. WFR ist eine Warteschlangee "Warte auf freien Rechner", in die jeder Prozeß eingetragen wird, der Bedienung erwartet. Diese Queue ist nach Prioritäten geordnet, der Inhalt ist ein Queue-Header, an den die Prozeß-Kontroll-Blöcke (PCB's s. 2.6.4.1) angekettet sind. Zu Beginn des Betriebes ist diese Queue leer. Das Einketten eines Prozesses in diese Queue ist gleichbedeutend mit dem Zustand 'bereit' eines Prozesses (s. 2.1.1). Die WFR ersetzt also die PROCESS_QUEUE's (s. Abb. 2-27) Bei jedem Zeitinterrupt schaut jeder Prozessor im Dual-Port RAM nach ob ein Anreiz vom IOP eingetragen wird. Falls ein Anreiz vorliegt, wird ein Anreiz-Aufnahme-Prozeß aktiviert und der Eintrag, als Meldung aufbereitet, in die dazugehörige Prozeß-Warteschlange eingetragen. Dann wird der alte Prozeß fortgesetzt, falls kein anderer höher priorisierter Prozeß an der WFR wartet. Der Anreiz-Aufnahmeprozeß hat so automatisch die höchste Priorität. Dieser geschilderte Vorgang der Abfrage des Dual-Port RAM und der eventuellen Prozeß-Verdrängung sieht in einem schematischen Struktogramm so aus (Abb. 6-10):

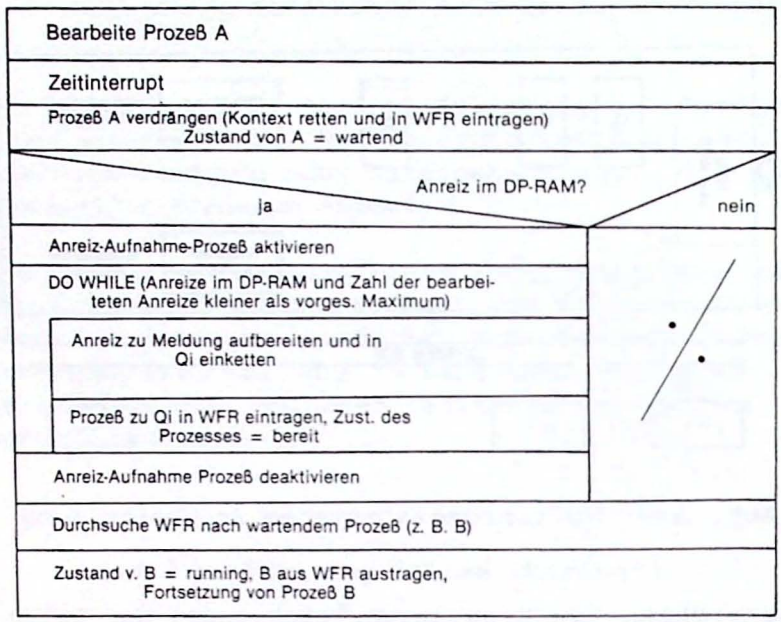

Abb. 6-10: Aufnahme von Anreizen bei Lastteilung

Ist kein Anreiz mehr aufzunehmen, da entweder das Dual-Port RAM leer ist, oder die maximale Zahl der Anreize abgearbeitet wurde, wird der nächst-priorisierte wartende Prozeß gesucht. So können z.B. durchaus mehrere Prozesse gleichzeitig Anreize aus dem Dual-Port RAM aufnehmen und im Anreiz-Aufnahme Prozeß weiterverarbeiten. Will man das zulassen, ist trotzdem höchste Vorsicht geboten, da durch unterschiedliche Verarbeitungsgeschwindigkeit der einzelnen Prozessoren sich Anreize auf dem Weg in die Prozeß-Warteschlangen überholen können (sog. "race conditions"). Dasselbe Problem tritt auf, wenn man zur Erhöhung der Parallelität in der Abarbeitung eines zeitaufwendigen Prozesses diesem Prozeß mehrere Eingangswarteschlangen zuordnen will. Lassen sich Gruppen von Meldungen logisch entkoppeln und entsprechend markieren und dadurch auf die richtigen Warteschlangen verteilen, ist das jedoch ein gangbarer Weg zur Erhöhung des Durchsatzes.

Bleiben wird der Einfachheit halber bei einer Eingangswarteschlange pro Prozeß. Der SEND-RECEIVE Mechanismus könnte folgendermaßen ablaufen (Abb. 6-11, Abb. 6-12). Die CASE-Verzweigung ist in Abb. 7-2 erklärt.

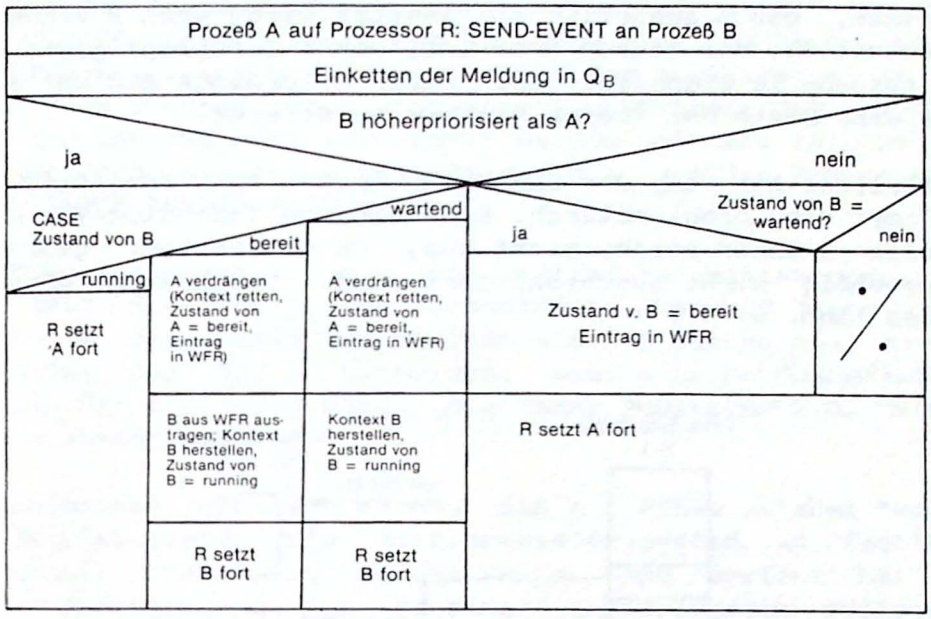

Abb. 6-11: SEND-EVENT bei Lastteilung

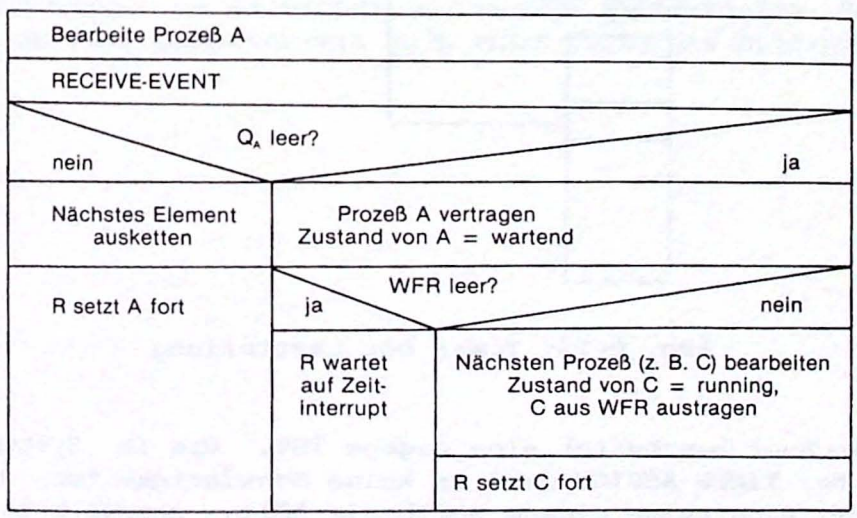

Abb. 6-12: RECEIVE-EVENT bei Lastteilung

Im rechten Zweig (B niedriger priorisiert als A) könnte man sich vorstellen, daß B schneller fortgesetzt wird, wenn R einem zweiten Rechner R2, der gerade frei ist, einen Interrupt gibt, um ihm eine Aufgabe zu signalisieren. Dafür allerdings muß im System-Memory eine Liste der freien Rechner existieren.

Auf ähnliche Art kann der SEMAPHORE Mechanismus realisiert werden. Timer sind problematisch. Eine einzige Time-Slot-Table (TST) für alle Rechner reicht nicht aus, da die Rechner gegenseitig hardwaremäßig nicht synchronisiert sind. Die Lösung könnte so aussehen (Abb. 6-13):

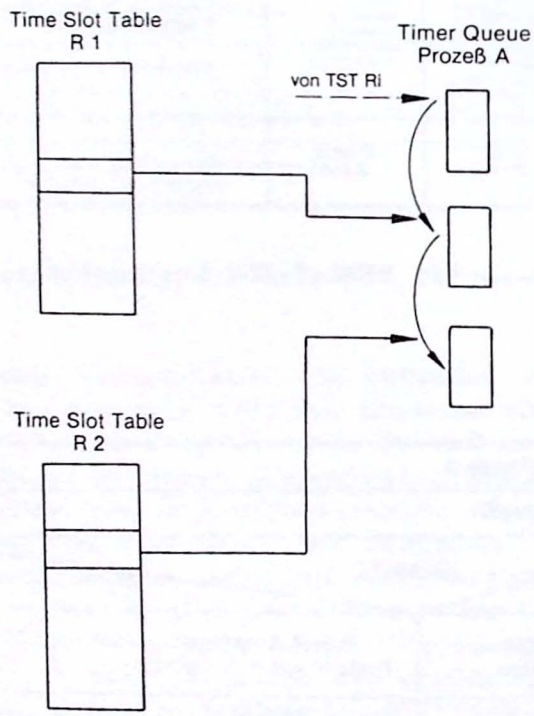

Abb. 6-13: Timer bei Lastteilung

Jeder Rechner bearbeitet eine eigene TST, die im System-Memory liegt. Bei TIMER_ASSIGN gibt es keine Schwierigkeiten, da jeder Prozessor sich seine eigene Zeitbasis hält, ebenso beim Ablauf eines Timers. Diese werden wie früher aus der (prozessor-eigenen) TST ausgekettet und verschickt. Einem Prozeß können Timer von

verschiedenen Prozessoren zugewiesen sein, die wie früher (s. 2.7.2.2) in der Timer-Queue verkettet sind. Damit das TIMER_RELEASE funktioniert, muß in jedes Timer-Poolstück die Prozessor-Namen eingetragen sein. Denn das Ausketten eines Poolstückes muß eventuell aus einer prozessorfremden TST geschehen. Daher muß die TST auch verriegelt werden und dazu ist im Poolstück der Verweis auf die TST nötig, d.h. auf den Prozessor, von dem das TIMER_ASSIGN erfolgte.

Ein Echtzeitkalender ist nur noch zentral zu realisieren, da auf lange Zeit die Prozessoren verschiedene absolute Zeiten zeigen würden. Im Betriebssystem MTOS68K z.B.ist hierzu ein Prozessor auszuwählen Das wäre zu vermeiden, wenn eine hardwaremässig realisierte Uhr vorhanden wäre, die jeder Prozessor z.B. im System Memory auslesen kann.

Zusammenfassend läßt sich sagen, daß mit einem solchen "verteilten" Betriebssystem ein Multiprozessor-System in Lastteilung wirkungsvoll realisiert werden kann. Man spricht bei einer solchen Realisierung von "software-transparentem Multiprocessing", da bei der Erstellung der Anwenderprozesse nicht berücksichtigt zu werden braucht, auf welchem der Prozessoren das Programm läuft.

Damit wollen wir den Abschnitt über Multiprocessing abschließen. Über verteilte Systeme in Lastteilung gibt es noch nicht sehr viel Erfahrung, da es bisher nur wenige kommerzielle Betriebssysteme für Mikrocomputer wie z.B. MTOS gibt, die Multiprocessing bieten.

Vierter Teil:

Softwareentwicklung für Anwenderprozesse

7. Softwareentwicklung für Anwenderprozesse in einer Realzeit-Multitasking-Betriebssystem-Umgebung

Dieses Kapitel behandelt praktische Probleme bei der Arbeit mit einem Realzeit-Betriebssystem. Für die hier beschriebene Technik der Zustands-Anreiz-Programmierung, die beschrieben wird, existiert die von CCITT genormte Sprache SDL (Specification and Description Language), die mit wenigen, einfachen, grafischen Symbolen einen überschaubaren Entwurf der Prozesse ermöglicht, der dann direkt, oft ohne den Zwischenschritt über Struktogramme, in die gewählte Programmiersprache umgesetzt werden kann. Innerhalb des gesamten Top-Down Software-Systementwurfes bildet dieser Schritt das Bindeglied zwischen verbaler Beschreibung und Zergliederung der Aufgabe einerseits und der eigentlichen Programmierung andererseits. Weiter wird auf die programmiertechnische Organisation der Prozesse, bis zur Abarbeitung einer empfangenen Meldung eingegangen.

7.1. Zustands-Anreiz-Programmierung

Die Schlagworte "Zustand" und "Anreiz" sind schon verschiedentlich verwendet worden, ohne daß auf Details eingegangen wurde. Diese Begriffe sollen verdeutlicht werden.

7.1.1 Zustände von Prozessen

Der Begriff 'Zustand' wird am ehesten an Beispielen klar. Als erstes Beispiel, das in diesem Abschnitt erläutert wird, dient ein Automat, mit dem Bankkunden nach Schalterschluß Geld abheben können. Das zweite Beispiel, erläutert in Abschn. 1.2, bezieht sich auf Abschnitt 6.2.2.1, und zwar auf den Prozeß INDIV_RENNEN.

1. Beispiel: Bankautomat

Wir kennen sie alle, die Automaten der Bankinstitute, mit denen auch nach Schalterschluß Geld vom Konto abgehoben werden kann. Der Vorgang ist im wesentlichen immer derselbe:

Durch Einschieben einer Berechtigungskarte (z.B. Scheckkarte) verschafft sich der Kunde die Benutzungsvollmacht des Automaten. Beispielsweise versperrt vor dem Einstecken der Scheckkarte ein Rolladen den Zugriff zur Tastatur. Nachdem also die Tastatur freigegeben ist, muß sich der Benutzer durch Eintippen einer nur ihm bekannten Nummer den Zugang zum Dienstangebot des Automaten verschaffen. Ist auch diese Hürde genommen, kann z.B. im Dialog mit dem System ein bestimmter Maximalbetrag abgehoben werden. Es wird Geld ausgegeben, die Karte zurückgegeben und nach einer Wartezeit der Rolladen wieder geschlossen.

Der Automat hat also mehrere Zustände, in denen er auf Eingaben von außen ganz verschieden reagiert (vgl. Abb 7-1).

1. Ruhezustand

Er ist dadurch gekennzeichnet, daß der Rolladen des Automaten geschlossen ist. Als Eingabe ist nur das Einschieben der Scheckkarte möglich.

2. Bereitschaftszustand

Der Vorgang eines Kunden ist komplett abgewickelt (oder noch nicht begonnen), der Rolladen ist oben, um einen eventuell wartenden weiteren Kunden sofort bedienen zu können.

Als Eingabe ist das Einschieben der Scheckkarte möglich. Auch die Tasten der Eingabetastatur sind zugänglich, allerdings ohne eine Reaktion hervorzurufen.

3. Prüfzustand 1

Die Kodierung des Magnetstreifens auf der Scheckkarte wird auf Gültigkeit geprüft, z.B. Bankinstitut oder Gültigkeitsdauer. Die Prüfung wird in Zusammenarbeit mit einer nachgeschalteten Datenverarbeitungsanlage (DVA) vorgenommen, in der sämtliche Kundendaten gespeichert sind.

4. Warten auf Geheimnummer

Vier Ziffern, die die Geheimnummer bilden, werden als Eingabe akzeptiert.

5. Prüfzustand 2

Die Geheimnummer wird auf Gültigkeit überprüft, z.B. ob sie mit der vom Magnetstreifen gelesenen Nummer übereinstimmt.

6. Warten auf Kundenwunsch

Vom Kunden wird im Dialog der Auftrag an den Automaten erteilt, also z.B. Kontostand abfragen, Geld abheben oder Geld einzahlen. Danach verzweigt das Programm, je nach Auftrag, in einen von mehreren möglichen Zuständen.

7. Auftragsabwicklung

Hierunter sind z.B. die Zustände Abheben, Einzahlen, Kontostand abfragen usw. zu verstehen, wobei nur einer der Zustände vorkommen kann.

8. Auftrag abschließen

Ausgabe der Scheckkarte und abwarten, ob ein neuer Auftrag angestoßen wird. Dies muß durch erneutes Einschieben einer Scheckkarte erfolgen.

9. Sperre

In diesem Zustand wird die Anlage gesperrt. Das kann in Fehlerfällen nötig sein, also z.B. wenn die übergeordnete DVA auf einen Prüfauftrag nicht antwortet.

Die Abb 7-1 gibt einen Überblick über die möglichen Prozeßzustände.

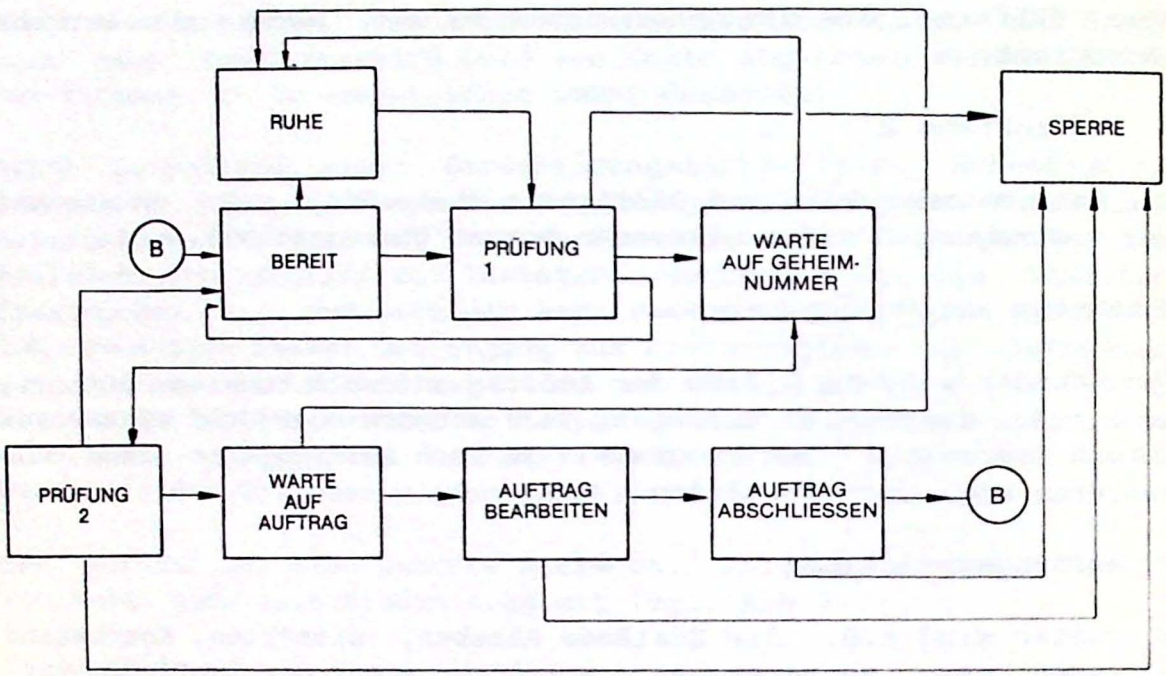

Abb. 7-1: Zustandsübergänge des Bankautomaten

Die Pfeile, die von einem Zustandskasten ausgehen, bedeuten mögliche, wenn auch im Bild nicht weiter beschriebene, Verzweigungen in andere Zustände. Man spricht von "Zustandsübergängen".

Der Zustand "BEREIT" hat zwei Ausgänge: in den Zustand "RUHE" dann, wenn nach Abwicklung eines Auftrages kein weiterer Auftrag gegeben wurde, und in den Zustand PRUEFUNG 1, wenn innerhalb dieser Wartezeit durch Einschieben einer Karte ein weiterer Auftrag erteilt wurde.

Falls die "PRUEFUNG 1" positiv verlief, geht es im Zustand "WARTE AUF GEHEIMNUMMER" weiter, andernfalls geht es in den "BEREIT"-Zustand zurück.

"PRUEFUNG 2", der nach Eingabe der Nummer eingenommen wird, hat drei Ausgänge: 1. Prüfung positiv, dann gehe nach "WARTE AUF AUFTRAG", 2. Geheimnummer bis zu zweimal fehlerhaft eingegeben, dann gehe nach "WARTE AUF GEHEIMNUMMER", und 3. Geheimnummer dreimal fehlerhaft, dann gehe nach "BEREIT".

Die drei übrigen Zustände laufen linear durch, wobei sich mit der Rückkehr nach "BEREIT", der Kreis wieder schließt. Von einigen Zuständen gehen Übergänge in den Zustand "SPERRE". Hier handelt es sich um einen Fehlerzustand, in dem unerwartete Fehlerfälle des Systems aufgefangen werden.

Insgesamt wird wieder deutlich: Die einzelnen Zustände bilden logische Trennlinien im gesamten Ablauf. Auch hier begegnet uns wieder, wie bei der Aufteilung der Aufgabe in einzelne Prozesse (s. 6.2.2), jetzt aber auf Prozessebene die Forderung, einen Ablauf in funktionale Teile zu zergliedern, nämlich in die einzelnen Zustände. Im gewählten Beispiel sind die einzelnen Zustände teils Abbild von unterschiedlichen "physischen" Zuständen des Gerätes (Rolladen oben oder unten), teils Ausdruck von erreichten Schritten im Gesamtablauf, z.B. "BEARBEITUNG ABSCHLIESSEN".

7.1.2 Anreize an Prozesse

Aus dem Beispiel wird klar, wie ein Zustandswechsel vor sich geht. In einem bestimmten Zustand können dem Prozeß eine Reihe von Anstößen von außen angeboten werden. Diese sind die sog. "Anreize" an den Prozeß, die ihm in Form von Meldungen, die über das Betriebssystem verteilt werden, zukommen. Anreize an den Prozeß des vorigen Beispiels sind Aktionen des Menschen, wie Einschieben der Scheckkarte oder Eingaben an der Tastatur. Diese "Hardware-Anreize" müssen von einem zwischengeschalteten Prozeß aufbereitet werden, um über das Betriebssystem als Meldung weitergeleitet werden zu können.

Hardware-Anreize sind jedoch nicht die einzigen Anreize, die unser Prozeß erhält. Anreize sind vom eigenen Betriebssystem möglich, z.B. wenn vom Prozeß angeforderte Zeitglieder ablaufen. Im Zustand "BEREIT" wartet der Prozeß eine bestimmte Zeit auf den Beginn eines neuen Auftrages. Falls die Zeit ohne neue Anforderung eines Bankkunden verstreicht, bekommt der Prozeß vom Betriebssystem Mitteilung und begibt sich in den Zustand "RUHE", nicht ohne vorher den Rolladen heruntergelassen zu haben. Weitere Anreize, die nicht von der Eingabe-Hardware erzeugt werden, sind z.B. Quittungen auf Prüfaufträge an eine nachgeschaltete Datenverarbeitungsanlage.

7.1.3 Folgerungen aus dem Beispiel

Die einzelnen Zustände eines Prozesses selektieren aus der Gesamtzahl der möglichen Anreize diejenigen heraus, die im gerade vorhandenen Zustand zu irgendwelchen Aktionen führen. Prozeßzustände können mit unmittelbar von außen erkennbaren Zuständen der Hardware verknüpft sein. Diese Formalisierung eines Prozeßablaufes paßt nahtlos in die Methode des modernen Software-Entwurfes:

1. Die Prozeßzustände müssen festliegen, bevor das Programm erstellt wird. Dadurch wird ein Zwang ausgeübt, sich den Ablauf eines Prozesses vorher im Detail zu durchdenken und durch die Zerlegung in einzelne Teilschritte, sprich Prozeßzustände, zu analysieren. Die Richtung ist dieselbe wie bei der Aufteilung eines Programmsystems in Prozesse: Zuerst analysieren, dann spezifizieren, dann programmieren.

2. In einem bestimmten Prozeßzustand muß davon ausgegangen werden, daß Anreize (Meldungen) an den Prozeß kommen, die eigentlich in diesem Zustand gar nicht vorgesehen sind. Zwei Fälle sind zu unterscheiden:

   a. Normale Betriebsfälle

   Beispiel: Im Prozeßzustand "PRUEFUNG 1" bedient der Kunde die Tastatur des Bankautomaten. Mit Fehlbedienungen muß immer gerechnet werden. Das Gerät sollte sich in diesen Situationen so tolerant verhalten, daß es die Eingabe entweder ignoriert oder besser, dem Benutzer Hinweise gibt wie "Bitte warten" oder "Eingabe noch nicht zugelassen".

   b. Fehlerfälle

   Beispiel: Im Prozeß "PRUEFUNG 1", in dem der Automat von einer nachgeschalteten DVA die Kontrolle der Scheckkarte wünscht, kommt überhaupt keine Antwort. Das ist kein normaler Betriebsfall, sondern ein Fehlerfall. Es kann sich, insbesondere in der Erprobungsphase um einen Programmierfehler handeln oder die DVA hat einen Defekt, kann also nicht quittieren. Der Prozeß "PRUEFUNG 1" darf nun keinesfalls auf die Prüfung warten, da diese wegen des Fehlers nie kommen wird. Der Effekt wäre, daß der Kunde frustriert weggeht, seine Scheckkarte einbehalten wird und der Automat

für weitere Kunden Bereitschaft signalisiert, die aber nicht vorhanden ist.

Der richtige Weg ist der, daß der Prüfauftrag an die DVA zeitüberwacht ist (z.B. 5 sec) und nach Ablauf der Kontrollzeit der Automat für eine weitere Benutzung gesperrt wird. Man spricht von "exception handling", d.h. kontrollierte Behandlung von nicht vorgesehenen Fehlerzuständen.

Der wichtigste Punkt ist, daß Prozesse aus allen Lebenslagen in definierte Zustände zurückkehren müssen. Alle am Geschehen Beteiligten, und zwar

- Benutzer (im Beispiel: Kunden des Bankautomaten)
- Betreiber (im Beispiel: die Bank)
- betroffene andere Prozesse des Automaten (im Beispiel etwa der Prozeß, der Scheckkarten annimmt oder ablehnt)

müssen in Fehlerfällen Hinweise bekommen, aus denen sie ihr weiteres Verhalten ableiten können. Für unser Beispiel würde das etwa bedeuten:

Benutzer: Anzeige, daß die Anlage vorübergehend nicht betriebsbereit ist, Rückgabe der Scheckkarte.

Betreiber: Möglichst detaillierte Angaben auf einer angeschlossenen Datensichtstation oder einem Drucker über das Fehlerbild, etwa: "DVA-Quittung ausgeblieben, Auftrag an DVA: Scheckkartenprüfung", möglichst mit Uhrzeit, Datum und soweit vorhanden, Angaben über die Scheckkarte.

Andere Prozesse: Meldungen die nötig sind, um den Automat für die weitere Benutzung zu sperren.

Der Punkt "Information an den Betreiber" enthält die ganze Problematik und Komplexität eines automatischen Überwachungskonzepts. Der Aufwand, den man hier treiben kann, ist fast beliebig hoch. Es ist immer ein Kompromiß zwischen Betreiberkomfort und Programmieraufwand zu suchen.

Ein weiterer Vorteil der Zustandsprogrammierung von Prozessen ist der, daß die Prozeßzustände üblicherweise in Tabellen festgehalten werden. Das heißt, Zustandsabfrage ist das Auslesen einer Tabelle und Zustandsübergänge sind Einträge in Tabellen. Insofern

ist die zustandsgesteuerte Programmierung sehr erweiterungsfreundlich in dem Sinn, daß neue Prozeßzustände leicht nachzubringen bzw. überflüssige leicht zu streichen sind, da Tabellenänderungen einfach vorgenommen werden können.

7.1.4 "Step Tables"

Kehren wir zu unserem Beispiel zurück:
Nehmen wir die Prozeßzustände "BEREIT" und "PRUEFUNG 1" und fassen in zwei Tabellen die möglichen Anreize und damit verbundenen Reaktionen für diesen Prozeß zusammen (Tab. 7-1 und 7-2).

Tabelle 7-1

Zustand "BEREIT"

ANREIZE	Tastatur-eingabe	Scheckkarte einschieben	Ablauf Timer 10sec
PROGRAMM-AKTIONEN	keine	Auftrag: Prüfung Magnetstreifen an DVA. --- Zeitauftrag 10sec zuweisen	Rolladen schließen
NEUER ZUSTAND	BEREIT	PRUEFUNG 1	RUHE

Tabelle 7-2

Zustand "PRUEFUNG 1"

ANREIZE	TASTATUR-EINGABE	QUITTUNG: PRUEFUNG OK	QUITTUNG: PRUEFUNG NEGATIV	ABLAUF TIMER 10 SEC	ABLAUF TIMER 5 SEC
PROGRAMM-AKTIONEN	keine	Ausgabe: GEHEIMNUMMER EINGEBEN	Scheckkarte einbehalten --- Ausgabe: KARTE UNGUELTIG --- Lösche 10 sec Timer --- Setze 5 sec Timer	Ausgabe: GERAET GESPERRT --- Ausgaben an Betreiber	Ausgabe: KARTE EINSCHIEBEN
NEUER ZUSTAND	PRUEFUNG 1	WARTE AUF GEHEIMNUMMER	PRUEFUNG 1	SPERRE	BEREIT

Diese Tabelle bildet einen Auszug aus der gesamten Zustands-Anreiz-Tabelle für den betrachteten Prozeß. In der Tabelle sind nur diejenigen Zustands-Anreiz-Kombinationen eingetragen, die sinnvolle Aktionen ergeben. Unsinnige Kombinationen müßten natürlich auch berücksichtigt werden und auf Fehlerzweige führen bzw. ignoriert werden. Beispielsweise muß im Zustand PRUEFUNG 1 der Versuch, eine Scheckkarte einzuschieben, ignoriert werden, d.h. der Motor des Kartenlesers darf nicht gestartet werden.

7.2. Grafische Hilfsmittel

Es gibt verschiedene grafische Hilfsmittel zur Software-Erstellung. Am meisten verbreitet sind die altbekannten Flußdiagramme, die auch in diesem Buch schon verschiedentlich zur Darstellung von allgemeinen Abläufen verwendet wurden. Sie sind zur Darstellung von Programmabläufen nicht sehr geeignet, da sie nicht die Regeln der strukturierten Programmierung unterstützen, in der z.B. jedes Unterprogramm genau einen Eingang und einen Ausgang hat.

7.2.1 Struktogramme

Die sog. "Struktogramme" oder "Nassi-Shneiderman-Diagramme" vermeiden diesen Nachteil. Wir wollen die strukturierte Programmierung hier als bekannt voraussetzen. Struktogramme sind ja im Buch auch schon ausgiebig verwendet worden. Lediglich auf die CASE-Verzweigung wollen wir nochmals etwas genauer eingehen.

In einer CASE-Verzweigung oder DO CASE-Abfrage wird je nach dem Wert des Arguments einer von verschiedenen Abläufen durchgeführt (Abb. 7-2).

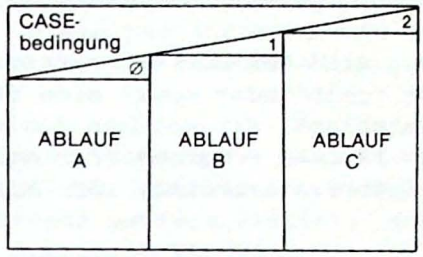

Abb. 7-2: "CASE"-Verzweigung

Programmsyntaktisch sieht das z.B. in PL/M so aus:

```
DO CASE bedingung;

     ablauf A;
     ablauf B;
     ablauf C;
END;
```

Hier wird entweder der Ablauf A, B oder C durchgeführt, je nachdem ob die Bedingung den Wert 0, 1 oder 2 hat.

Diese einfache Form der CASE-Verzweigung hat verschiedene Nachteile, weswegen wir auch noch das entsprechende Sprachelement in der Programmiersprache CHILL erwähnen. Die Nachteile in PL/M sind:

1. Der Bedingung muß explizit eine Zahl zugewiesen werden. Es ist nicht erlaubt, "bedingung" einen symbolischen Wert zuzuweisen.

2. Es gibt keine mehrdimensionale CASE-Verzweigung.

3. Es gibt keinen ELSE-Zweig, der eine Programmverzweigung darstellt, wenn "bedingung" unerwartete Werte annimmt.

In CHILL (und ähnlich in PASCAL oder ADA) werden diese Nachteile vermieden. Zuerst kann der Zahlenvorrat, den "bedingung" annehmen kann, bei der Deklaration der Variablen "bedingung" symbolisch erklärt werden:

```
DCL bedingung = SET (rot, gelb, grün, blau, lila);
```

Mit dieser Erklärung wird der Wertevorrat von "bedingung" als entweder "rot" oder "gelb" oder sonst eine der Farben aus der Menge (engl. SET) festgelegt. Mit welchen Zahlen der Compiler rot usw. darstellt, ist für den Programmierer nebensächlich, da er nur noch mit den Symbolen arbeitet. In CHILL sähe die CASE-Verzweigung so aus:

```
CASE bedingung OF

     (rot):    ablauf_A;
     (grün):   ablauf_C;
     (gelb):   ablauf_B;
     (ELSE):   ablauf_sonst;
ESAC;
```

Man sieht, die Elemente des CASE müssen nicht in der Reihenfolge der Definition im SET angegeben werden. Im ELSE-Zweig sind alle Abläufe zusammengefaßt, die nicht für rot, gelb, oder grün zutreffen, also wenn der Wert von "bedingung" "blau" oder "lila" ist.

Der Grund, warum wir genauer auf die CASE-Verzweigung eingehen, ist der, weil sich damit auf sehr einfache Art Zustandsanreizverzweigungen realisieren lassen, wie später noch gezeigt wird (s. 7.3.2.2).

Die Vorteile von Nassi-Shneiderman Diagrammen sind:

1. Die einzelnen Strukturelemente lassen sich in modernen Programmiersprachen eindeutig zur Sprachsyntax zuordnen. Diese Sprachen nennen sich aus diesem Grunde "strukturiert". Beispiele sind PL/M, PASCAL, ADA, CHILL. Nicht strukturierte Sprachen sind z.B. FORTRAN, BASIC, ASSEMBLER. Durch die leichte Übersetzbarkeit von Struktogrammen in den Code einer strukturierten Sprache, wird die Zeit für die eigentliche Programmierung stark reduziert.

2. Die Regeln der strukturierten Programmierung sind einfach. Ihre Anwendung führt zu klar gegliederten, überschaubaren Programmen. Es ist möglich, genaue Detaillierung von einzelnen Programmteilen später in den

Entwurf nachzubringen. Durch die Vorschrift des Zusammenfügens einzelner Struktogrammelemente werden im Programm unübersichtliche Querbeziehungen schon im Entwurf "verboten".

Nachteile gibt es natürlich auch. Hauptsächlich stört, daß bei vielen Schachtelungen sehr schnell der Platz auf dem Papier knapp wird und daher meistens mehrere Blätter, auch zur Darstellung einfacher Abläufe, gebraucht werden.

Struktogramme eignen sich gut zur Darstellung konventioneller, nacheinander ablaufender Programme und Programmteile. Ungeeignet sind sie als Hilfsmittel zur Entwicklung und Veranschaulichung parallel ablaufender Prozesse, die unter der Regie eines Multitasking-Betriebssystems ablaufen.

Hierfür gibt es ein einfaches und dem Zweck angemessenes Hilfsmittel, die sog. Zustands-Anreiz-Diagramme, oder "SDL"-Diagramme.

7.2.2 Zustands-Anreiz-Diagramme ("SDL"-Diagramme)

SDL bedeutet "Specification and Description Language" und ist vom International Telegraph and Telephone Consultant Commitee (CCITT) im Document AP VII-No 20-E im Juni 1980 veröffentlicht worden. SDL war ursprünglich als Hilfsmittel zur Beschreibung der "internen logischen Prozesse" von rechnergesteuerten Telefonvermittlungssystemen geschaffen worden . Die Methode ist jedoch so allgemein und einfach, daß sie unserer Meinung nach hervorragend zur Spezifikation und Beschreibung von parallel ablaufenden Prozessen im Sinne des Prozeßkonzepts geeignet ist.

Wir werden hier keine wörtliche Übersetzung des zitierten CCITT-Dokuments wiedergeben, da die Formulierung ziemlich verklausuliert ist. Vielmehr werden wir die SDL-Symbole anhand der jetzt vertrauten Begriffe Zustand, Anreiz, Meldung usw. erklären.

7.2.2.1 SDL-Symbole

Die folgende Tabelle 7-3 gibt einen Überblick über die wichtigsten SDL-Symbole:

Tabelle 7-3

Symbol	Erläuterung
	Zustandssymbol: Ein Prozeß wartet in diesem Zustand auf einen Anreiz, er hat sich z.B. mit RECEIVE_EVENT in diesem Zustand vertagt. In das Symbol wird der Zustand eingetragen.
	Anreiz: In das Symbol wird das bestimmte Ereignis eingetragen, das den Prozeß aus dem Zustand, in dem er sich vertagt hat, aufweckt. Dem hier eintreffenden Ereignis muß im selben oder einem anderen Prozeß ein Auslöser entsprechen, der den Anreiz bewirkt. Normalerweise ist das eine mit SEND_EVENT abgesandte Meldung. In einem Zustand kann natürlich auf viele Ereignisse gewartet werden, die dann in der Reihenfolge ihres Eintreffens in der Eingangswarteschlange abgearbeitet werden.
	Meldung: In das Symbol wird die Meldung eingetragen, die im Normalfall mit SEND_EVENT an einen anderen Prozeß abgesandt wird und dort in einem bestimmten Zustand als Anreiz eintrifft.
	Entscheidung: Die Raute hat dieselbe Bedeutung wie in den konventionellen Flußdiagrammen. Es wird eine Verzweigung im Prozeßablauf durchgeführt.
	Ablauf: Das Symbol hat ebenfalls dieselbe Bedeutung wie bei den Flußdiagrammen. Es wird ein Ablauf durchgeführt, der einen Eingang und einen Ausgang hat, in sich aber beliebig kompliziert sein kann.

Der Ablauf wird üblicherweise durch ein Struktogramm detailliert wiedergegeben.

Verzweigung: Sie eintspricht der CASE-Verzweigung der Struktogramme.

Konnektor: Hier wird angedeutet, daß das SDL-Diagramm bei dem Zeichen, das im Konnektor eingetragen ist, fortgesetzt wird, da an der jetzigen Stelle der Platz zum Zeichnen zu eng wird.

Diese Symbole bilden die Grundlage von SDL. Sie werden ähnlich wie die bekannten Flußdiagramme, nämlich im zeitlichen Ablauf von oben nach unten, auf dem Papier gezeichnet. Zum Zusammenfügen der Symbole sind noch einige einfache Regeln zu beachten, die aber beim Anwenden des Prozeßkonzeptes fast selbstverständlich sind.

Ein Prozeß, der zyklisch durchlaufen wird, verhält sich immer nach demselben Schema: Er vertagt sich in einem bestimmten Zustand z.B. im RECEIVE_EVENT, d.h. er wartet in diesem Zustand auf eines oder mehrere Ereignisse (Anreize), die ihm über das Betriebssystem von einem anderen Prozeß mit SEND_EVENT übermittelt werden und ihn aus seiner Ruhe aufwecken. Ein Durchlauf des Prozesses muß daher, von SDL gesehen, mit einem Zustandssymbol beginnen, auf das Anreizsymbole folgen müssen (Abb. 7-3).

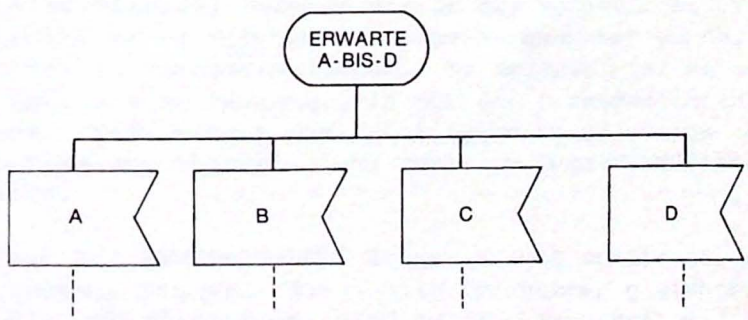

Abb. 7-3: Zustands-Anreiz-Verzweigung in SDL

Der Prozeß hat sich im Zustand ERWARTE_A_BIS_D vertagt und kann nur durch eines der Ereignisse A...D aufgeweckt werden. Es wird also in diesem Zustand entweder gewartet oder eines der Ereignisse A, B, C oder D ausgeführt.

Eine Ausnahmesituation tritt ein, wenn er in diesem Zustand einen anderen Anreiz bekommt, auf den er eigentlich nicht gewartet hat. Dann wird er entweder eine Fehlerroutine durchlaufen, oder das Ereignis wird ignoriert. Diese Fälle werden üblicherweise nicht in den SDL-Diagrammen gezeichnet, sondern im Rahmenprogramm des Prozesses abgefangen (s. 7.3.2). Im Verlauf der Abarbeitung eines erlaubten Anreizes wird der Prozess verschiedene Aktionen durchführen, bevor er sich wieder vertagt, und zwar entweder in demselben oder einem anderen Zustand. Ein Zweig eines SDL-Diagrammes endet also auch in einem Zustandssymbol.

7.2.2.2 Beispiele

Wir wollen nun an Hand einiger Beispiele betrachten, wie SDL-Diagramme in der Praxis eingesetzt werden. Gehen wir zurück zu unserem Beispiel aus Abschnitt 7.1.4, (Tabelle 7-1 und 7-2), dem Bankautomat. Dort hatten wir Zustands-Anreiz Verküpfungen schon in Form von Tabellen aufgestellt, die wir nun ohne weiteres als SDL-Diagramme darstellen können (Abb. 7-4).

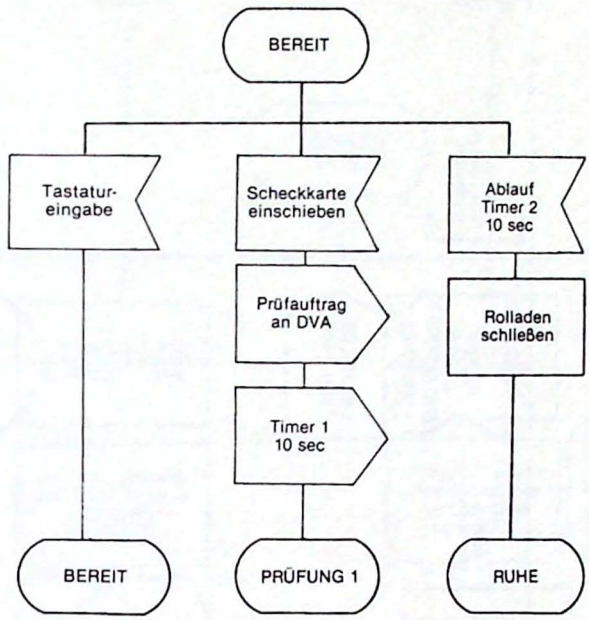

Abb. 7-4: SDL-Diagramm für Zustand "BEREIT" des Bankautomaten

Es ist nun eine triviale Übung, die Tabelle 7-2 ebenfalls als SDL- Diagramm zu übertragen.

Als nächstes Beispiel knüpfen wir an das Kapitel. 6.2.2 an, wo eine Kontrolle eines Sportwettkampfes hergenommen wurde, um das Prozeßkonzept zu veranschaulichen. Im Beispiel ist es notwendig, sich auf Abb. 6-3 zu beziehen, in der die Prozeßaufteilung dargestellt ist. Wir werden die SDL-Diagramme von zweien der dort erwähnten Prozesse angeben, und zwar vom Prozeß UEBERWACHUNG und INDIV_RENNEN.

Beginnen wir mit UEBERWACHUNG: Dieser Prozeß sollte alles regeln, was das gesamte Rennen, d.h. alle Teilnehmer gleichermaßen betrifft. Die SDL-Diagramme sind in den Abb. 7-5 und Abb. 7-6 dargestellt. Die Pfeile mit der Beschriftung PROC_A...PROC_E beziehen sich auf die programmtechnische Realisierung. Diese wird in Kapitel 7.3.2 angesprochen.

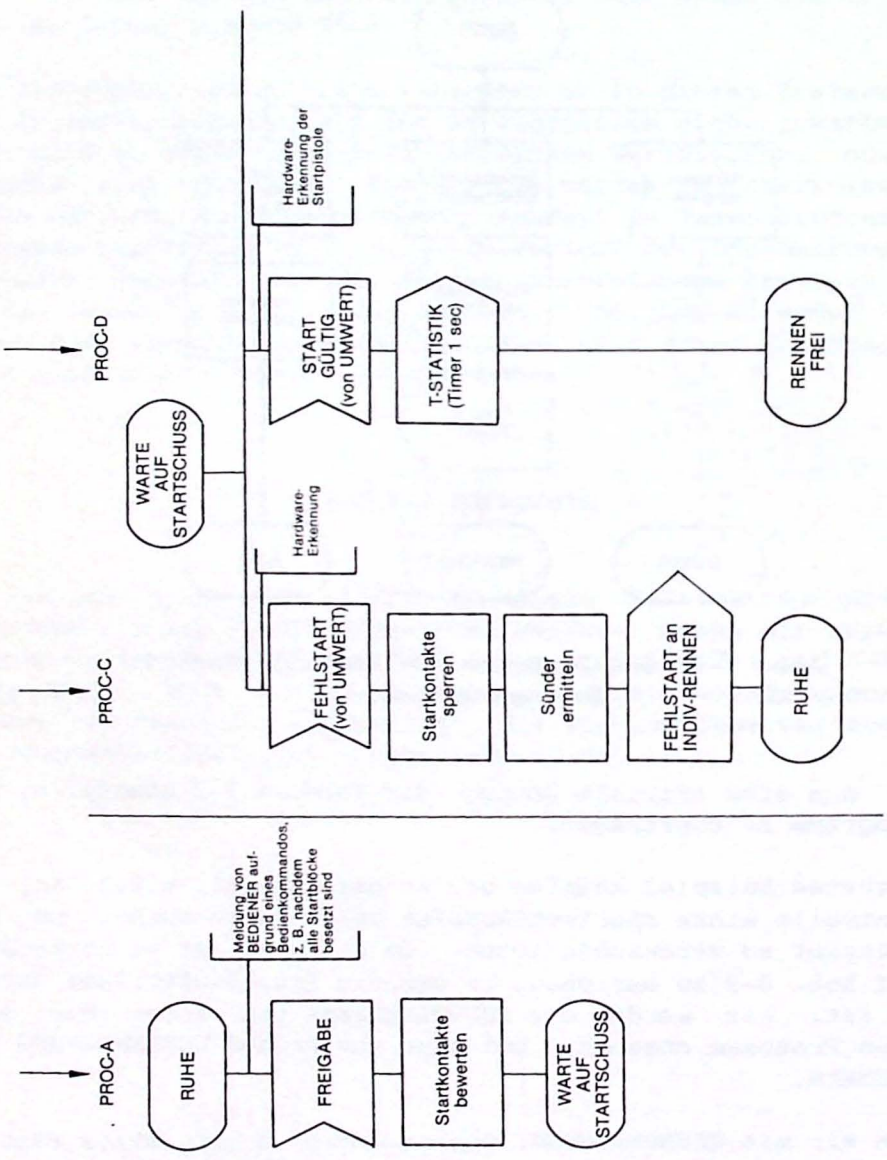

Abb. 7-5: SDL-Diagramme von Prozeß-Überwachung

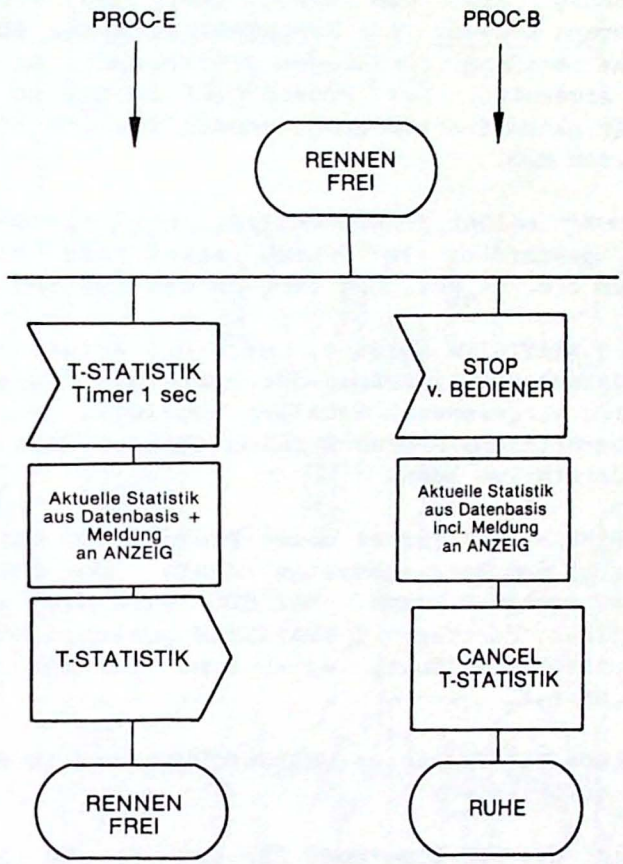

Abb. 7-6: Prozeß-Überwachung

Der Prozeß wartet nach dem Hochlauf, der kein Bestandteil der SDL-Diagramme ist, im Zustand RUHE auf ein einziges Ereignis, nämlich das Bedienkommando FREIGABE, das beim Schwimmen gegeben wird, wenn alle Teilnehmer auf den Startblöcken stehen. Der Ablauf "Alle Startkontakte bewerten" gibt Aufträge an die Hardware, daß von jetzt ab das Verlassen der Startblöcke (beim Schwimmen) als Start gewertet wird (als guter Start oder Fehlstart). Danach wird ein Zustandsübergang vorgenommen, der Prozeß vertagt sich im Zustand WARTE AUF STARTSCHUSS.

In diesem Zustand sind zwei Anreize möglich, die beide von einer Hardware-Erkennung auf dem Umweg über den Prozeß UMWERT (s 6.2.2.1) kommen müssen. Bei FEHLSTART wird der Sünder ermittelt und eine Meldung für diesen Teilnehmer an den Prozeß INDIV_RENNEN abgesetzt. Der Prozeß geht zurück in den Zustand RUHE, da ja der ganze Startvorgang, einschließlich Startfreigabe, wiederholt werden muß.

Falls die Hardware meldet START GUELTIG, wird mit dem Startschuß die Stoppuhr gestartet. Im Prozeß setzt man ein Zeitglied T_STATISTIK von z.B. 1 sec, und geht in den Zustand RENNEN FREI.

Jedesmal wenn T_STATISTIK abläuft, wird die aktuelle Rennstatistik aus den Datenbasis-Einträgen der einzelnen Teilnehmer gebildet und in dafür vorgesehenen Tabellen abgelegt, wo sie sich der hier nicht beschriebene Prozeß ANZEIGE abholen kann und auf der Anzeigetafel darstellen kann.

Im Zustand RENNEN FREI wartet unser Prozeß also auf den Anreiz T_STATISTIK, der vom Betriebssystem kommt, oder auf den Anreiz STOP, der vom Bediener kommt. Bei STOP wird die abschließende Statistik gebildet, der Timer T_STATISTIK gestoppt und der Prozeß in den Ruhezustand überführt, wo er z.B. auf den Start eines neuen Rennens wartet.

Der Programmcode des Prozesses UEBERWACHUNG ist im Anhang 2 als PL/M Programm aufgelistet.

Abb. 7-7 zeigt die SDL-Diagramme für den Prozeß INDIV_RENNEN.

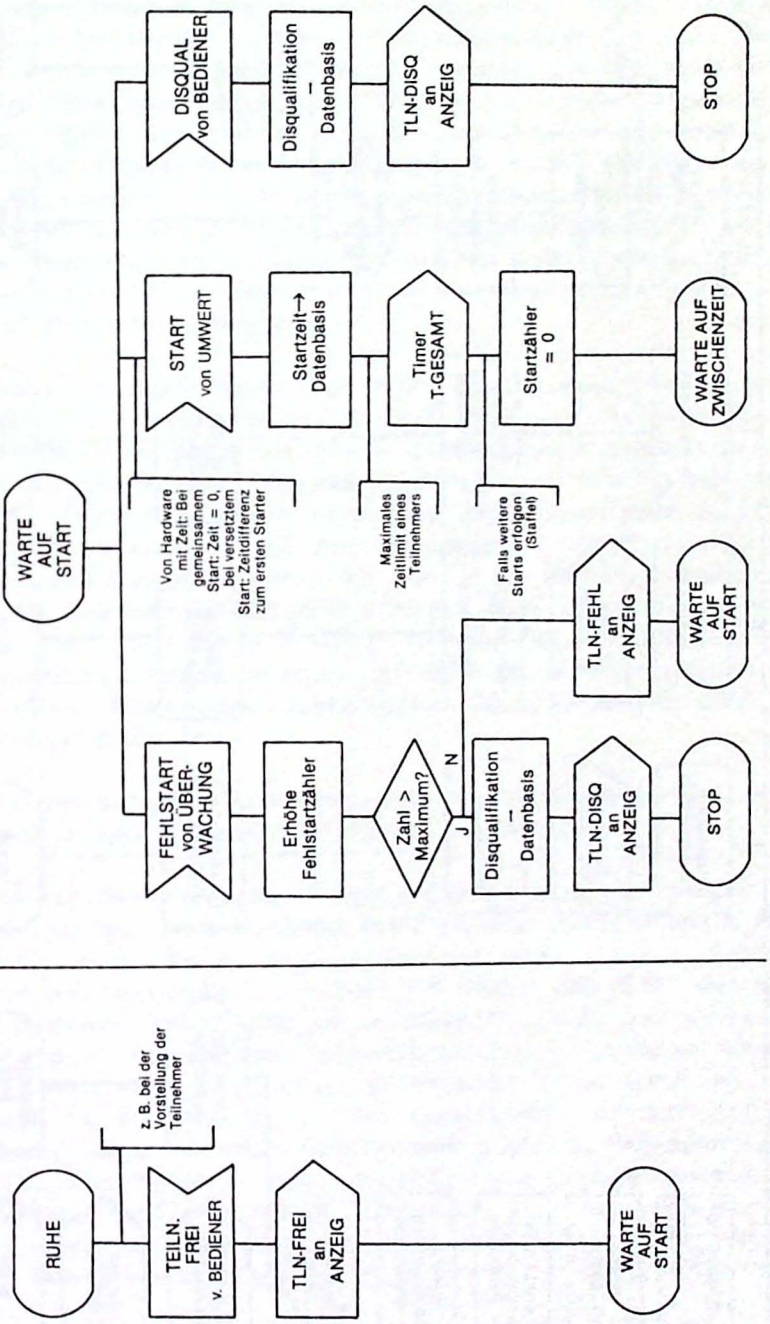

Abb. 7-7/1: INDIV_RENNEN

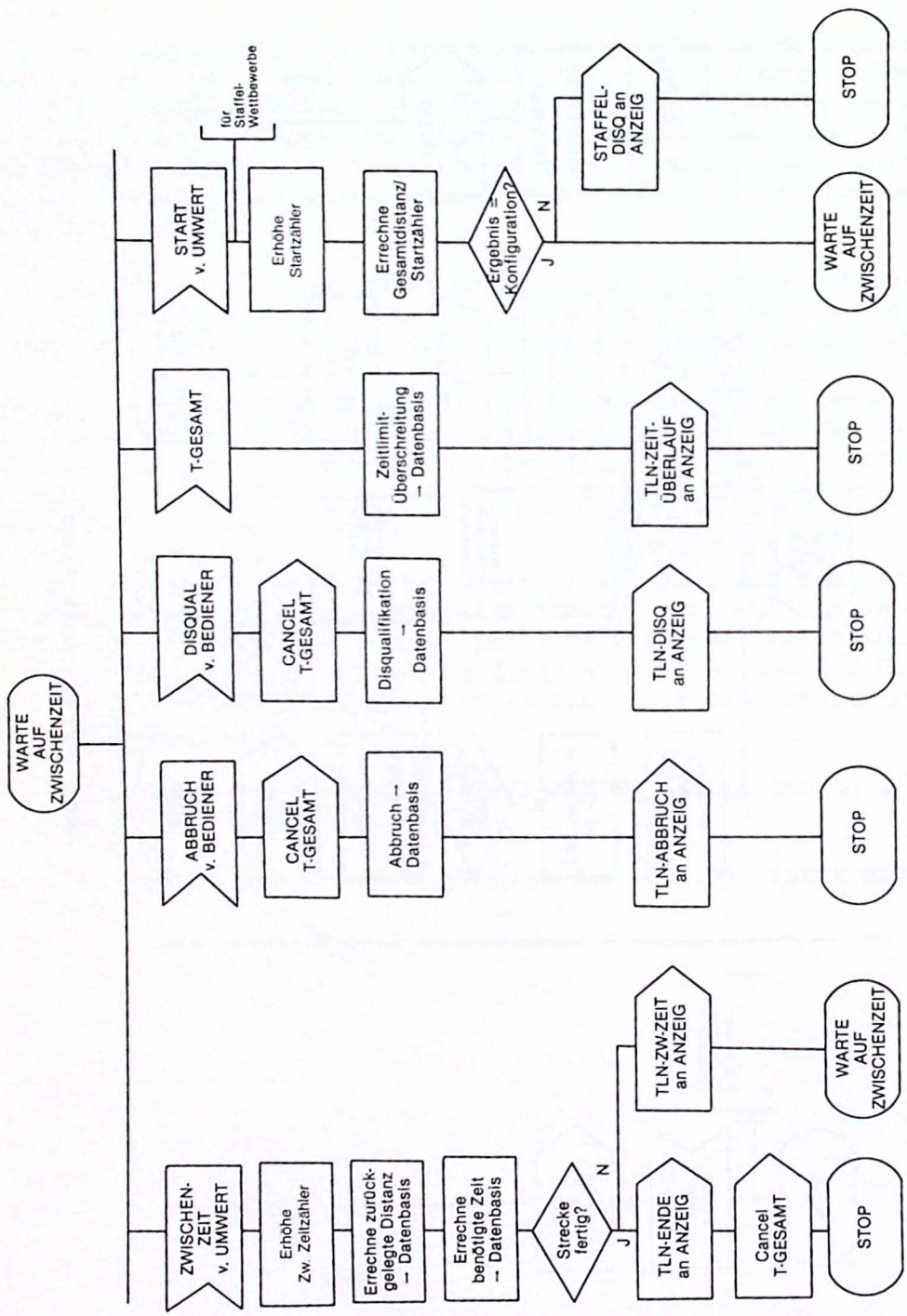

Abb. 7-7/2: INDIV_RENNEN

Auch dieser Prozeß beginnt mit dem Zustand RUHE, aber für einen bestimmten Teilnehmer. Das Bedienkommando TLN_FREI wird z.B. gegeben, wenn der betreffende Teilnehmer dem Publikum vorgestellt wird. Im Zustand WARTE_AUF_START kann ein ordnungsgemäßer Start ablaufen oder ein Fehlstart oder der Teilnehmer kann auch vom Bediener per Eingabekommando disqualifiziert werden, wenn er z.B. den Schiedsrichter beschimpft. Bei erfolgreichem Beginn des Rennens geht er in den Zustand WARTE_AUF_ZWISCHENZEIT, in dem alle für das Rennen wichtigen Ereignisse aufgenommen und bewertet werden. Die Details der einzelnen Zweige durchzugehen, soll als Übung für den Leser bleiben.

Auf einen wichtigen Punkt sei hier hingewiesen: Sendet ein Prozeß einem anderen eine Meldung, so muß im Ablauf sichergestellt sein, daß diese Meldung auch in einem sinnvollen Prozeßzustand ankommt. Im Beispiel sendet der Prozeß UEBERWACHUNG dem Prozeß INDIV_RENNEN nach einem Fehlstart eine Meldung. Zu diesem Zeitpunkt muß sichergestellt sein, daß der Teilnehmer, dem diese Meldung gilt, auch in dem Zustand ist, in dem diese Meldung bewertet wird, nämlich im Zustand WARTE_AUF_START. Hier begegnet uns wieder das Problem, daß solche oft nicht einfach zu überschauende Fälle in aller Ausführlichkeit vorher auf dem Papier durchgespielt werden sollen. Die Übersichtlichkeit von SDL-Diagrammen trägt dazu in wesentlichem Maße bei.

Einige Bemerkungen, bezogen auf die teilnehmerindividuelle Organisation eines Prozesses, seien angebracht:

Im Beispiel des Prozesses INDIV_RENNEN wird ein Teilnehmer nach dem anderen aus dem Zustand RUHE in den Zustand WARTE AUF START überführt. Ein Teil der Teilnehmer kann also im Zustand RUHE sein, ein weiterer Teil im Zustand WARTE_AUF_START und wieder ein Teil im Zustand WARTE_AUF_ZWISCHENZEIT, z.B. bei einer Sportart, bei der die Teilnehmer einzeln oder in Gruppen nacheinander starten. Man kann also hier nicht mehr davon sprechen, daß sich der Prozeß im Zustand A, B oder C befindet, sondern die einzelnen Teilnehmer, die ja alle zum Prozeß gehören, befinden sich in den einzelnen Zuständen. Der Prozeß, wie er als Einheit vom Betriebssystem bedient wird, ist also hier nur noch ein formaler Überbau, der aus Gründen der effizienten Ausnutzung der Rechner-Hardware beibehalten wird.

Demzufolge müssen auch die "Prozeß"- Zustände teilnehmerindividuell geführt werden. Dies ist kein Problem da z.B. mit der Teilnehmernummer als Index in ein Array aus Datenstrukturen adressiert werden kann, von denen jede aus den teilnehmerindividuellen Daten besteht. Jeder Meldung wird als Kennzeichnung zusätzlich die Teilnehmernummer mitgegeben, damit der Teilnehmer vom Prozeß identifiziert und seine eigene Teilnehmertabelle gefunden werden kann.

7.3. Die Organisation von Prozessen

Prozesse, die einmal vom Betriebssystem gestartet sind, können verschiedene Verläufe nehmen. Sie können linear vom Anfang bis zum Ende durchlaufen und dann z.B. vom Betriebssystem wieder gelöscht werden (DELETE_PROCESS, S. 2.1.2). Ein gelöschter Prozeß schafft im Hauptspeicher Platz für andere, nichtresidente Prozesse, die etwa von einem Plattenspeicher nachgeladen werden können. Solche Prozesse werden selten Realzeitaufgaben erledigen, da die Realzeitfähigkeit des Systems meist ständig erhalten bleiben muß und zeitkritische Prozesse permanent verfügbar sein müssen.

Solche ständig ablaufbereiten Prozesse wird man meist als zyklische Prozesse realisieren. Nach dem Prozeß-Start durch das Betriebssystem wird ein Initialisierungsteil durchlaufen. Danach mündet der Prozeß in eine Endlosschleife, aus der er nicht mehr herauskommt und in der er seine möglichen Kombinationen von Zuständen und Anreizen abarbeitet.

7.3.1 Prozeß-Hochlauf

Im Prozeß-Hochlauf wird alles das erledigt, was der Prozeß später zum ordnungsgemäßen Abarbeiten seiner Endlosschleife braucht. Es handelt sich beim Hochlauf meist um ein linear durchlaufendes Programmstück, in dem etwa Zustandstabellen oder Warteschlangen initialisiert werden. Überwachungsprozesse können hier z.B. erste Diagnosen und Kontrollen durchführen, beispielsweise ob alle notwendigen Betriebsmittel wie externe Baugruppen, Massenspeicher usw. angeschlossen sind und richtig arbeiten. Ebenso ist denkbar,

daß sich verschiedene Prozesse im Hochlauf aufeinander synchronisieren, d.h. immer stückweise weiterlaufen, bis vom Partnerprozeß eine Meldung kommt. Die Einzelheiten hängen ganz vom jeweiligen Anwendungsfall ab. Prozesse, die im Hochlauf kritische Phasen durchlaufen, sollten beim Erreichen von wichtigen Zwischenzuständen irgendwelche Marken setzen, um dem Benutzer auch nach außen hin anzuzeigen, daß der Hochlauf bis hierher erfolgreich war. Ob das durch Ansteuern von Displayanzeigen wie Leuchtdioden passiert oder durch Ausgabe an einer Datensichtstation oder sonstwie wird von Fall zu Fall verschieden sein. Hier trifft das in Kap. 5.2.2 Gesagte zu, wonach schon sehr einfache Hardware-Anzeigen in Fehlerfällen wertvolle Hinweise geben können.

7.3.2 Die Organisation der Endlosschleife

In diesem Abschnitt werden wir eine übliche Form der Organisation eines zyklischen Prozesses darstellen der sich mit RECEIVE_EVENT vertagt hat und darauf wartet, daß ihm ein anderer Prozeß mit SEND_EVENT eine Meldung schickt und ihn dadurch aufweckt. Die hier gezeigte Form hat sich wegen der Einfachheit und Übersichtlichkeit bewährt. Es ist natürlich nicht die einzig mögliche Form.

Der Ablauf der Endlosschleife teilt sich in zwei Teile auf (Abb. 7-8):

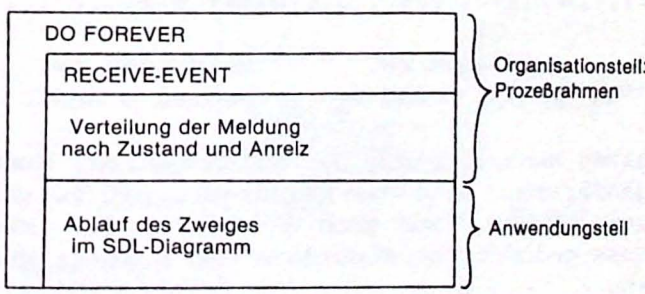

Abb. 7-8: Ablauf der Endlosschleife

Es gibt den inneren Teil, der die Abarbeitung eines Zweiges des Zustands-Anreiz-Diagrammes darstellt und den äußeren Teil, der dafür sorgt, daß dieser innere Zweig richtig angesteuert und erreicht wird. Von diesem äußeren Teil spricht man auch als "Prozeßrahmen". Mit anderen Worten, ein Prozeß teilt sich in einen Organisationsteil, den Prozeßrahmen, und einen Anwendungsteil, der die programmtechnische Realisierung des Zustands-Anreiz-Diagrammes darstellt.

7.3.2.1 Die Klassifikation von Meldungen

Meldungen werden als Poolstücke verschickt. Im mit RECEIVE_EVENT empfangenen Poolstück muß also die komplette Information eingetragen sein, die der Empfänger zur Interpretation der Meldung braucht. Diese Information ist von der zu unterscheiden, die das Betriebssystem zum Ein- und Ausketten in den Warteschlangen braucht, nämlich FORWARD_LINK und PID (=Process-Identification, s. 2.3.1.2). Diese Parameter sind dem Anwender wohl zugänglich, man sollte aber vermeiden, damit zu arbeiten, um vom Anwenderprogramm gegen eventuelle Änderungen des Betriebssystems, die sich z.B. auf die PID beziehen, geschützt zu sein.

Wie klassifiziert man praktischerweise Meldungen?

Die Gesamtheit der Meldungen, die es in einem System gibt, kann von 0 bis n durchnumeriert werden. Das könnte dann so aussehen:

0,1,2,.....,k,k+1,k+2,.....,m,m+1,m+2,.....,n

Meldungen zw. Prozeß A,B	Meldungen zw. Prozeß C und D	Meldungen zw. Prozeß B und C

Bei der gewählten Kennzeichnung der Meldungen mit Nummern ist der Nachteil augenfällig: Soll der Meldungsvorrat zwischen Prozeß A und B erweitert werden, muß auch der Vorrat der Meldungen der anderen Prozesse geändert werden, da A und B jetzt Nummern größer als k brauchen.

Erfolgt die Kennzeichnung der Meldungen symbolisch durch Elemente eines SET (s 7.2.1), ist dieser Nachteil zunächst vermieden, da der Compiler das "Nachrücken" automatisch besorgt. Sind beim SET

jedoch Meldungen z.B. an einen externen Prozessor dabei, dessen Programm in PROMs abgelegt ist, muß auch dessen Programm neu compiliert werden, auch wenn sich die Änderung nicht auf den Teil des SET bezieht, der die Schnittstelle zum externen Prozessor betrifft.

Daher ist ein anderer Weg vorzuziehen:
Die Meldungen zwischen je zwei Prozessen werden in einem kleinen Zahlenvorrat bzw. SET festgelegt (JOBCODE). Mit einer zweiten Zahl bzw. SET wird das Prozeßpaar gekennzeichnet, um dessen Schnittstelle es sich handelt (EVENT_TYPE):

JOBCODE	0.....k	0.....p	0.....q	
Prozesse	A <-> B	C <-> D	B <-> C	, zusätzlich

EVENT_TYPE	0	1	2

Eine Meldung ist nun also eindeutig durch JOBCODE und EVENT_TYPE klassifiziert. Die oben geschilderten Probleme treten jetzt höchstens auf, wenn der EVENT_TYPE Vorrat erweitert wird. Das wird jedoch bei weitem weniger häufig der Fall sein als beim JOBCODE. Durch die Aufspaltung in JOBCODE und EVENT_TYPE wird das Prinzip des "information hiding" (s. 6.2.2) gewahrt. Änderungen bleiben auf die absolut notwendigen Teile beschränkt.

Unser Poolstück könnte jetzt also so aussehen, wie in Abb. 7-9 gezeigt.

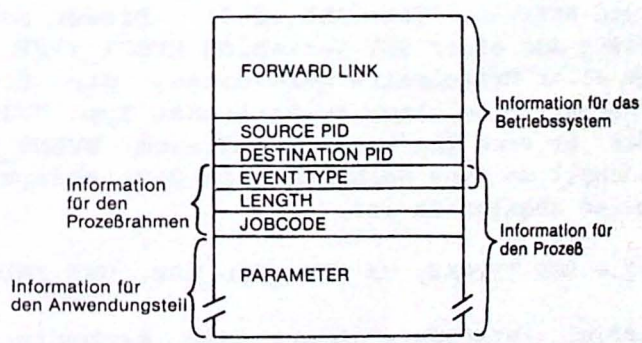

Abb. 7-9: Aufbau eines Poolstücks

Der Parameter LENGTH ist oft nützlich, z.B. wenn die Meldung über ein Datenlink (s. 1.1.4) an einen fremden Rechner verschickt werden soll. Dann braucht nicht das ganze Poolstück, sondern nur der tatsächlich benötigte Teil übertragen zu werden. Der dreigeteilte Aufbau des Poolstückes zeigt also ganz klar die Abgrenzung der Aufgabenbereiche:

1. Information für das BS: Wird nur vom BS, nicht vom Anwender benutzt.

2. Information für den Prozeßrahmen: Wird vom "Organisationsteil" des Prozesse, dem "Prozeßrahmen", benutzt, um den eingetroffenen Anreiz entsprechend dem Prozeß-Zustand an den Anwendungsteil zu verteilen.

3. Information für den Anwendungsteil: Wird innerhalb eines Zweiges des Zustands-Anreiz Diagrammes benutzt, um den dort erforderlichen Ablauf korrekt durchzuführen.

7.3.2.2 Verteilung der Anreize an die Anwenderprogramme

Wir wollen nun das Struktogramm von Abb. 7-7 verfeinern, und zwar den Kasten "Verteilung der Meldung nach Zustand und Anreiz".

Am besten greifen wir wieder auf unser altes Beispiel aus Kapitel 6.2.2 zurück und betrachten den Prozeß UEBERWACHUNG. Der Prozeß hat Schnittstellen zu den Prozessen UMWERT, BEDIENER, INDIV_RENNEN und ANZEIGE. (s. Abb .6-3). Diesen Schnittstellen seien EVENT_TYPES aus einer SET-Variablen EVENT_TYPE zugeordnet, die die Menge aller Prozeßpaare bezeichnen, die Schnittstellen miteinander haben, plus einem zusätzlichen Typ "TIMER". TIMER zeigt an, daß es sich bei einer mit diesem EVENT_TYPE ausgestatteten Meldung, um eine Nachricht vom Betriebssystem handelt, daß ein Zeitglied abgelaufen ist.

```
DCL EVENT_TYPE = SET (TIMER, UM_UEB, BED_UEB, UEB_INDIV,....,);
```

Für das Beispiel interessieren nur die Elemente UM_UEB und BED_UEB, entsprechend den Schnittstellen UMWERT -> UEBERWACHUNG bzw. BEDIENER -> UEBERWACHUNG.

Die Schnittstelle UM_UEB meldet die Ereignisse FEHLSTART und START_GUELTIG. Über die Schnittstelle BED_UEB werden die Meldungen FREIGABE und STOP gemeldet. Für die entsprechenden JOBCODES definieren wir wieder zwei SET-Variablen.:

```
DCL UM_UEB_JOBCODES  = SET (FEHLSTART, GUTSTART);
DCL BED_UEB_JOBCODES = SET (TLN_FREI, STOP);
```

Diese Datendeklarationen sind in einer Form, wie sie der Sprache CHILL entsprechen, sähen jedoch in PASCAL oder ADA ganz ähnlich aus.

Die letzte noch ausstehende Information, die der Prozeßrahmen zur Verzweigung benötigt, ist die SET-Variable für die Zustände der einzelnen Prozesse. Für den Prozeß UEBERWACHUNG ergibt das

```
DCL UEB_STATES = SET (RUHE, WARTE, FREI);
```

Die Elemente des SET entsprechen den Zuständen RUHE,
WARTE AUF START, RENNEN FREI.(s. Abb. 7-5)

Das Struktogramm für den Prozeßrahmen sieht dann folgendermaßen aus (s. Abb. 7-10):

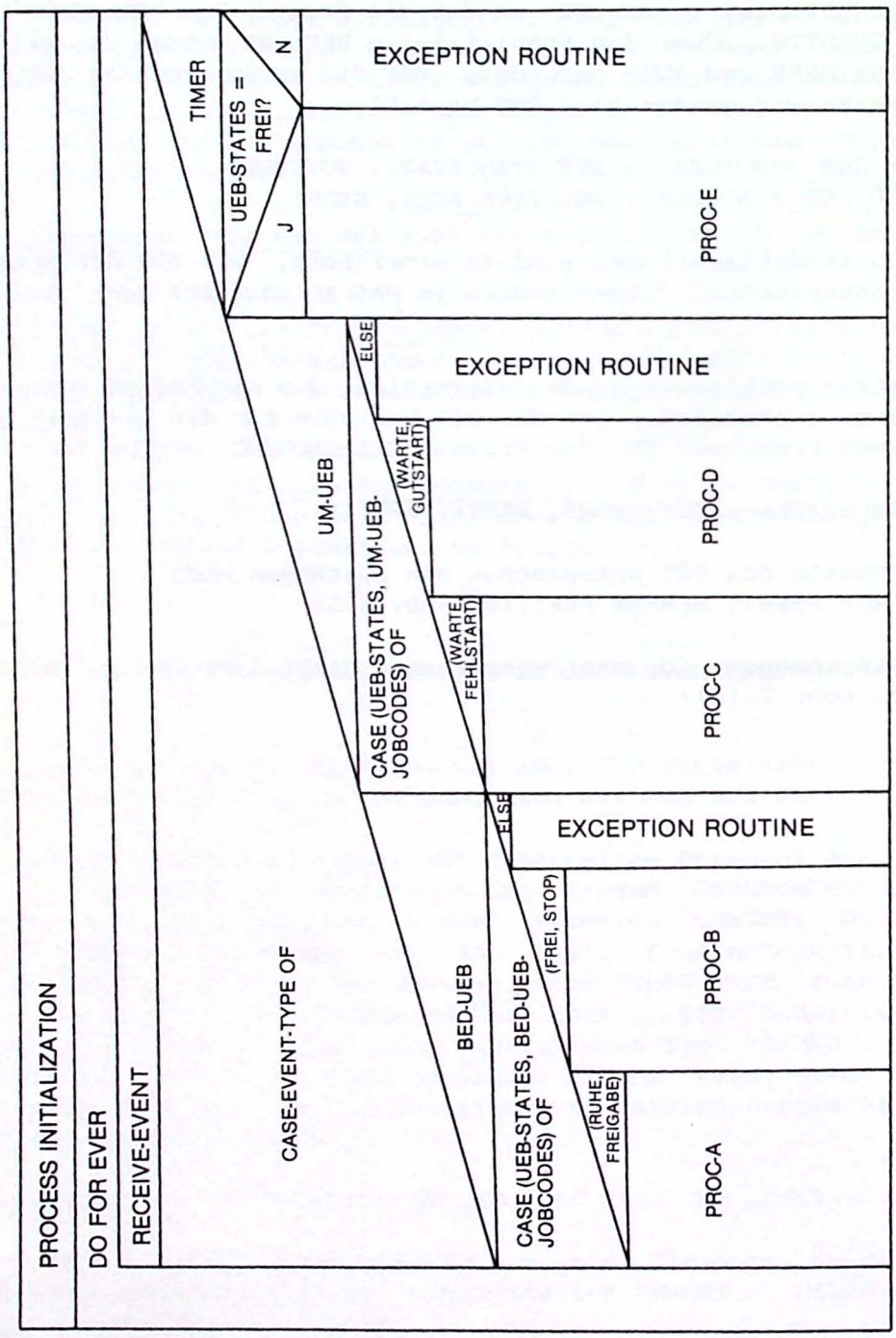

Abb. 7-10: Prozeßrahmen von ÜBERWACHUNG

Die Prozeduren PROC_A bis PROC_E sind entsprechend den Vorgaben des Zustands-Anreiz-Diagrammes zu programmieren. Dabei darf nicht vergessen werden, das empfangene Poolstück am Ende mit RELEASE_POOL wieder freizugeben, falls es nicht vorher weiterverwendet wurde, z.B. um im durchlaufenen Zweig einen SEND_EVENT abzusetzen.

Die aktuellen Bezugsgrößen EVENT_TYPE und JOBCODE beziehen sich natürlich immer auf das momentan bearbeitete Poolstück. Die abkürzende Zeile im Struktogramm "CASE EVENT_TYPE OF" sieht im tatsächlichen Code (hier: CHILL) etwas anders aus. Angenommen, das Poolstück wird über eine Zeigervariable EVENT_POINTER adressiert, die auf ein Datenfeld (das Poolstück) zeigt, das so aufgebaut ist, daß EVENT_TYPE ein Strukturelement dieses Datenfeldes ist. Die Verzweigung hieße im CHILL-Code so:

```
CASE EVENT_POINTER->.EVENT_TYPE OF
   (UM_UEB): ......
      .
      .
```

Ebenso wäre möglich:

```
DO WITH EVENT_POINTER->;
   CASE EVENT_TYPE OF

      (UM_UEB):.......
         .
         .
```

Daher haben wir in Abb. 7-10 diese abgekürzte Form gewählt. In der Sprache PL/M ist das ganze Beispiel im Anhang 2 als Programm beschrieben. Dieses PL/M-Programm zeigt an einem praktischen Beispiel die Realisierung der genannten Prinzipien der Organisation von Prozessen. Es gibt außerdem ein Beispiel, wie die Betriebssystemaufrufe (hier von 'ROSY') in der Praxis eingesetzt werden. Die anwendungsspezifischen Abläufe sind als externe Prozeduren angegeben (z.B. FEHLSTART_VERURSACHER). Die internen Details dieser Prozeduren sind für die hier verfolgte Absicht irrelevant, könnten aber jederzeit als separater Modul bei der Generierung eines ablauffähigen Programmsystems miteingebunden werden. Auf ein Detail in der Prozedur PROC_C sei noch hingewiesen:

Normalerweise muß vor dem SEND_EVENT mit dem Aufruf GET_POOL zuerst ein Poolstück vom Betriebssystem angefordert werden. Darauf kann hier verzichtet werden, da dem Prozeß vom RECEIVE_EVENT her ja schon ein Poolstück zur Verfügung steht, was zum Zeitpunkt des SEND_EVENT vollständig abgearbeitet ist und daher weiterverwendet werden kann.

Dieses Beispiel des Sportwettkampfes hat sich durch die Kapitel 6 und 7 wie ein roter Faden hindurchgezogen. Die näher beschriebenen Stationen des Top-Down Entwurfes dieses Beispiels waren:

1. Aufteilung der Aufgabe in einzelne Prozesse: Abb. 6-3.

2. SDL-Diagramme von UEBERWACHUNG und INDIV_RENNEN: Abb. 7-5, 7-6 und Abb. 7-7.

3. Struktogramm des Prozeßrahmens von UEBERWACHUNG: Abb. 7-10.

4. Auflistung des codierten Programms von UEBERWACHUNG: Anhang 1.

Das Beispiel UEBERWACHUNG wurde gewählt, da es wegen seiner Einfachheit leicht zu überschauen ist. Die gezeigten Entwurfsprinzipien und Programmiermethoden sind jedoch auch für wesentlich komplexere Systeme dieselben. Werden sie konsequent befolgt, ist das Ergebnis ein Programmsystem, das die Schlagworte "modular", "überschaubar", "gut wartbar" wirklich verdient.

In den abschließenden Abschnitten wollen wir nun noch auf einige Besonderheiten der Prozeßorganisation eingehen.

7.3.3 Teilnehmerindividuelle Meldungen

Wie in Abschnitt 6.2.2.1 erläutert, war der Prozeß INDIV_RENNEN aus dem Beispiel des Sportwettkampfes teilnehmerindividuell zu organisieren. Jedem Teilnehmer war sein individueller Prozeßzustand zuzuordnen, wobei die Abläufe für alle Teilnehmer aus technischen Gründen in einen einzigen Prozeß eingebettet waren. Diese Teilnehmerindividualität muß natürlich auch im Prozeßrahmen berücksichtigt werden. Das bedingt, daß sich diese Eigenschaft in dem Aufbau der Poolstücke niederschlägt, da der

Prozeßrahmen ja aufgrund der ihm dort gelieferten Information verzweigt. Das kann im Beispiel einfach so geschehen, daß im Poolstück ein oder mehrere Bytes "Teilnehmer-Nummer" zusätzlich zu der in Abb. 7-9 gezeigten Information vorgesehen werden. Über diese Teilnehmernummer verzweigt dann der Prozeßrahmen zusätzlich zu EVENT_TYPE, Zustand und JOBCODE. Der Zustand kann jetzt nicht mehr einfach eine SET-Variable sein, also global für den ganzen Prozeß gelten, sondern muß ebenfalls teilnehmerindividuell ausgewiesen sein. Das geschieht am besten in einer Tabelle, die mit der Teilnehmernummer indiziert wird (ARRAY). Die ganze Änderung zum obigen Prozeßrahmen besteht darin, daß statt der Verzweigung

```
CASE (ZUSTAND, EREIGNIS) OF
```

die Abhängigkeit

```
CASE (ZUSTAND (TEILNEHMER), EREIGNIS) OF
```

verwendet wird.

7.3.4 Priorisierung von Meldungen

Bei der üblichen Anwendung des SEND-RECEIVE-Mechanismus zur Prozeßkommunikation werden alle Meldungen, die ein bestimmter Prozeß erhält, gleichberechtigt behandelt. Das heißt, die Meldung, die zeitlich zuerst in der Eingangswarteschlange eintrifft, wird auch zuerst vom Zielprozeß abgehandelt. Es gibt aber Fälle, in denen gewisse Meldungen priorisiert werden müssen.

7.3.4.1 Prozessinterne Meldungen

Meldungen, die ein Prozeß an sich selbst schickt, können verschiedene Ursachen haben. Beispielsweise wäre es im obigen Beispiel des Prozesses INDIV_RENNEN denkbar, daß die Teilnehmer gegenseitige Abhängigkeiten haben und daher Meldungen innerhalb des Prozesses zwischen verschiedenen Teilnehmern ausgetauscht werden müssen. Allgemein können prozeßinterne Meldungen nötig werden, wenn der Prozeß aus mehreren Komplexen besteht, die lose

Abhängigkeiten zueinander aufweisen und sich daher durch Meldungen verständigen. Um die Übersichtlichkeit über zeitliche Abhängigkeiten zu steigern, ist es sinnvoll, diese prozeßinternen Meldungen zu priorisieren. Das heißt, daß der Prozeß kein neues Element von anderen Prozessen bearbeitet, bis alle internen Meldungen abgearbeitet sind.

Oft ist es ausreichend, maximal eine interne Meldung pro Prozeßdurchlauf zu betrachten. Das bedeutet nicht, daß es nicht viele verschiedenen interne Meldungen geben kann. Vielmehr heißt es, daß in einem Durchlauf der Endlosschleife nur eine einzige interne Meldung anfallen kann.

Die interne Meldung könnte ganz normal mit SEND_EVENT an den eigenen Prozeß geschickt werden, allerdings wäre sie dann nicht priorisiert. Um das Ziel der Priorisierung zu erreichen, kann man das Poolstück für die interne Meldung besonders kennzeichnen, z.B. indem man es an eine eigens definierte Zeigervariable anhängt, das heißt, die Variable (sie könnte INTERN_POINTER heißen) mit der Adresse des zugehörigen Poolstücks beschreibt. Die Abarbeitung geht nun einfach so vor sich, daß im Prozeßrahmen als erstes abgefragt wird, ob am INTERN_POINTER ein Poolstück hängt, also eine Internmeldung ansteht. Falls ja, wird sie bearbeitet und der INTERN_POINTER anschließend gelöscht, falls nein wird die Prozeßeingangswarteschlange mit RECEIVE_EVENT abgefragt. Dieser Vorgang sieht im Struktogramm folgendermaßen aus (Abb. 7-11)

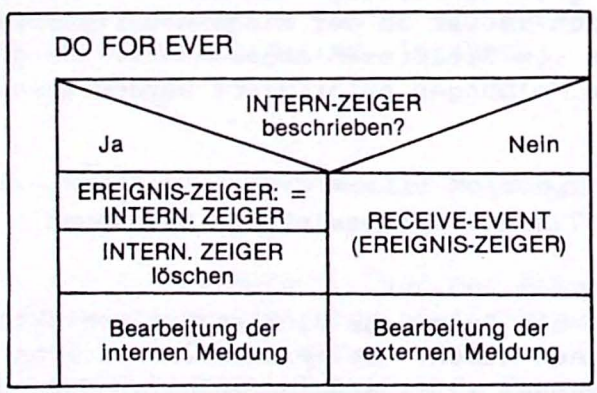

Abb. 7-11: Prozeßinterne Meldungen

Mit dieser einfachen Zusatzabfrage ist erstens die Priorisierung der Internmeldung erreicht worden und zweitens ist Bearbeitungsgeschwindigkeit gewonnen worden, da kein zeitaufwendiger Betriebssystemaufruf notwendig ist, sondern die notwendige Verwaltung vom eigenen Prozeßrahmen erledigt wird.

Können mehrere internen Meldungen pro Prozeßdurchlauf entstehen, muß man die Poolstücke in eine besondere interne Warteschlange ketten. Der Prozeßrahmen fragt dann, ob die interne Warteschlange leer ist. Falls nicht, wird das nächste Poolstück ausgekettet und bearbeitet. Das Struktogramm ist in Abb. 7-12 gezeigt.

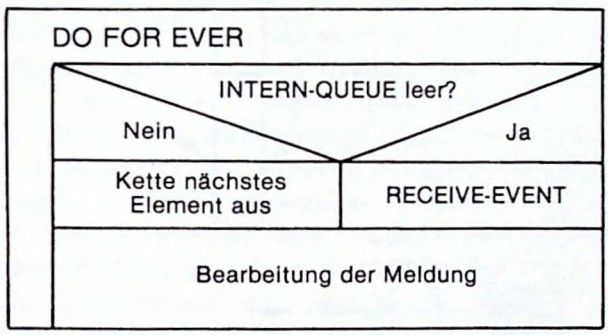

Abb. 7-12: Interne Warteschlange

7.3.4.2 Priorisierung von Meldungen zwischen verschiedenen Prozessen

Handelt es sich um Meldungen zwischen verschiedenen Prozessen, könnte man natürlich denselben Mechanismus verwenden. Davon ist aber strikt abzuraten, da das prozeßübergreifende Prioritätsschema durch die Prozeßprioritäten gegeben ist und nicht durch Sondermaßnahmen durchbrochen werden sollte, um die Überschaubarkeit der Abläufe nicht zu erschweren. Ist der Fall trotzdem nicht zu vermeiden, sollten keine Privatkonstruktionen zur Priorisierung verwendet werden, da es sich um prozeßübergreifende Aktionen handelt. Für diesen Fall bieten manche Betriebssysteme spezielle SEND-Aufrufe, z.B. SEND_EVENT_TOP genannt, die das abgesandte Poolstück ganz oben in die Warteschlange des Zielprozesses einkettet (s. Abb. 7-13).

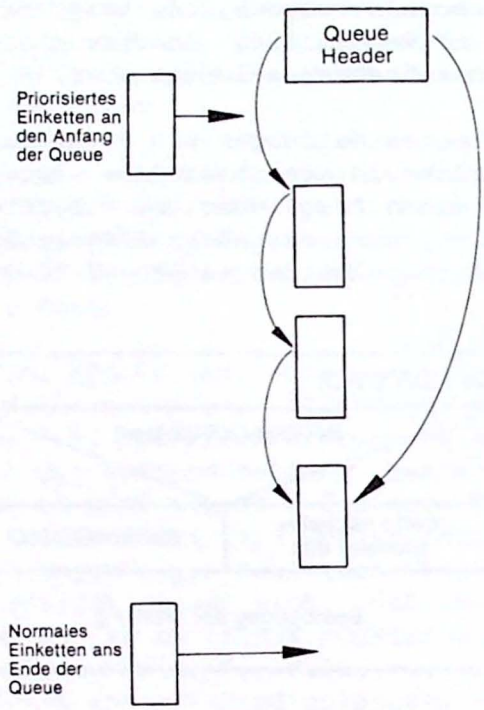

Abb. 7-13: Priorisiertes Einketten von Meldungen

Werden mehrere Poolstücke auf diese Weise verschickt, muß man beachten, daß die zeitliche Reihenfolge der Abarbeitung umgekehrt wird. Die Anwendung dieses Hilfsmittels wird umso problematischer, je mehr Anwender es benutzen wollen. Setzt z.B. ein Prozeß einen SEND_EVENT_TOP ab, und wird im weiteren Verlauf von einem höher-priorisierten Prozeß unterbrochen, der ebenfalls einen SEND_EVENT_TOP an denselben Zielprozeß sendet, wird wegen der Prioritätsstruktur der erste SEND_EVENT_TOP im Zielprozeß nicht zuerst behandelt. Die Priorisierung von Meldungen zwischen verschiedenen Prozessen sollte also möglichst ganz vermieden werden.

In den SDL-Diagrammen werden interne Meldungen beim Senden bzw. als Anreiz beim Empfangen mit den Symbolen von Abb. 7-14 bezeichnet.

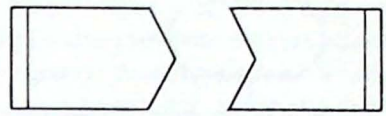

Abb. 7-14: SDL-Symbole für Internmeldungen

7.3.5 Verzögern von Meldungen

Es gibt Fälle, wo ein Prozeß Meldungen bekommt, die er zwar nicht ignorieren darf, aber zum Zeitpunkt des Empfanges noch nicht bearbeiten kann, weil er dazu noch auf ein anderes Ereignis warten muß. Er muß die Meldungen also solange aufheben, bis besagtes Ereignis eintrifft. Eine einfache Lösung ist wieder die, daß die Meldungen in eine besondere Warteschlange eingetragen werden, die dann zu gegebener Zeit abgearbeitet wird. Beim Verzögern oder "Vertagen" von Meldungen darf nicht der Fehler gemacht werden, daß sich der Prozeß die Meldung, die er für später aufheben will, immer wieder selbst schickt. In diesem Fall würde sich der Prozeß nie vertagen da die Meldung immer in der Eingangswarteschlange hängt. Niedrig-priorisierte Prozesse würden blockiert werden, das System arbeitet nicht mehr einwandfrei. Für Meldungen, die verzögert werden sollen, existiert in SDL ein eigenes Symbol (Abb. 7-15).

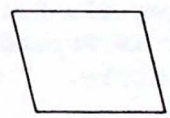

Abb. 7-15: SDL-Symbol für Verzögern von Meldungen

In einem Zweig eines Zustandsanreiz-Diagrammes sieht das Verzögern einer Meldung dann so aus (Abb. 7-16):

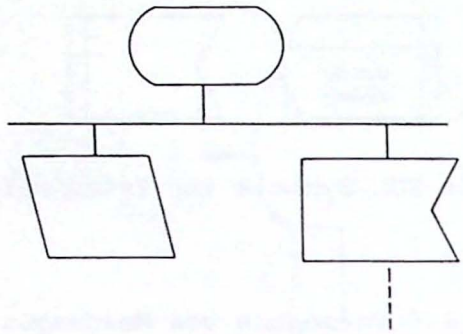

Abb. 7-16: Verzögern von Meldungen

7.3.6 Verwaltung von Prozeß-Zuständen

Für die Anreize an Prozesse, also die Meldungen, die zwischen Prozessen verschickt werden, müssen bestimmte Regeln eingehalten werden, da Meldungen prozeßübergreifende Datenstrukturen sind, die deswegen auch in einem Systen allgemeiner Festlegung bedürfen. Wir hatten als Beispiel die Klassifizierung nach JOBCODES und EVENT_TYPES genannt. Prozeßzustände dagegen sind private Eigenschaften der einzelnen Prozesse. Daher kann im Grunde jeder Autor eines Prozesses selbst festlegen, wie er seine Prozeßzustände kennzeichnet. Ein Prozeßzustand könnte z.B. dadurch markiert sein, daß eine besonders definierte Warteschlange nicht leer ist, oder daß ein Hardware-Port einen bestimmten Wert hat. Nimmt man diese Abfragen als Kriterium, müssen diese eventuell komplizierten Abfragen im Prozeßrahmen durchgeführt werden, um den Prozeßzustand zu ermitteln.

Sinnvoller ist es jedoch, diese Arbeit nicht dem Prozeßrahmen aufzubürden, sondern dem Ablauf im entsprechenden Zweig des Zustands-Anreiz-Diagrammes, um den Prozeßrahmen einfach zu halten. Der Prozeßzustand wird nämlich sowohl vom "Organisationsteil" des Prozesses, also dem Rahmen, als auch vom "Anwendungsteil" des Prozesses, also dem durchlaufenen Zweig des Zustands-Anreiz-Diagrammes benutzt. Daher ist es anzuraten, möglichst klare Datenstrukturen für die Charakterisierung des Zustandes zu wählen und komplizierte Abfragen oder Abläufe zu vermeiden. Man kann z.B. für die prozeßspezifischen Daten, die sich im Ablauf ändern,

eine Tabelle benutzen, von der ein Teil der Prozeßzustand ist. Das Anwendungsprogramm hat dann die Aufgabe, den neuen Prozeßzustand auf möglicherweise komplizierte Art zu ermitteln, und bildet diesen Vorgang durch einen einfachen Eintrag in eine Tabelle ab, um den Zustand zu kennzeichnen. So ist für den Prozeßrahmen zur Verzweigung für den Zustand lediglich dieser Eintrag zu lesen. Da, von Spezialfällen abgesehen, bei jedem Durchlauf des Prozesses der Zustand gelesen werden muß, sich dieser Zustand aber nicht bei jedem Durchlauf ändert, hat man so auch einen Laufzeitvorteil gewonnen.

Auch hier spiegelt sich das Prinzip des "information hiding" wider, nämlich daß eine Softwareschnittstelle auf das notwendige Minimum beschränkt bleibt. Die Schnittstelle, um die es sich hier handelt, ist die zwischen Prozeßrahmen und Anwendungsteil des Prozesses. Dem Prozeßrahmen bleibt die Information verborgen, welche Abfragen und Abläufe zur Bildung des Prozeßzustandes notwendig sind.

Anhang 1:

Compilerlisting des Betriebssystem-Kerns »ROSY« (Real Time Operating System)

```
/*****************************************************************
**                                                              **
**   PROZEDURNAME :  QUEUE                                      **
**                                                              **
**   FUNKTION     :  EINKETTEN EINES NEUEN ELEMENT AN DAS ENDE EINER QUEUE **
**                                                              **
**   PARAMETER    :  - QHDR_ADR : ADRESSE DER QUEUE             **
**                   - NEW_ADR  : ADRESSE DES EINZUKETTENDEN ELEMENTS **
**                                                              **
**   LAUFZEIT     :  155 uS - 177 uS                            **
**                                                              **
*****************************************************************/

102   1     QUEUE: PROCEDURE (QHDR_ADR, NEW_ADR) REENTRANT PUBLIC;

103   2      DECLARE QHDR_ADR        POINTER,
                      NEW_ADR        POINTER;
104   2      DECLARE QHDR            BASED QHDR_ADR QUEUE_HEADER_STRUCT,
                      NEW_ELEMENT    BASED NEW_ADR  GENERAL_POOL_ELEMENT_STRUCT;
105   2      DECLARE QUE_ELE_ADR     POINTER,
                      QUEUE_ELEMENT  BASED QUE_ELE_ADR GENERAL_POOL_ELEMENT_STRUCT;


106   2      IF QHDR.NUM_IN_QUEUE = 0
107   2         THEN QHDR.FIRST_IN_QUEUE = NEW_ADR;
108   3         ELSE DO; QUE_ELE_ADR = QHDR.LAST_IN_QUEUE;
110   3                  QUEUE_ELEMENT.FWD_LINK = NEW_ADR;
111   3               END;
112   2      QHDR.LAST_IN_QUEUE      = NEW_ADR;
113   2      QHDR.NUM_IN_QUEUE       = QHDR.NUM_IN_QUEUE + 1;
114   2      NEW_ELEMENT.FWD_LINK    = NULL_ADDRESS;


115   2     END QUEUE;

          $EJECT
```

```
SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE ROSY
OBJECT MODULE PLACED IN :F5:ROSY.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:ROSY.PLM


           $DEBUG
           $LARGE
           $OPTIMIZE(2)
           $PAGELENGTH (70)
           $PAGEWIDTH(131)
           $TITLE ('    R E A L T I M E - O P E R A T I N G - S Y S T E M   ')

           /*****************************************************************************
           ******************************************************************************
           **                                                                          **
           **          RRRRRR            00            SSSS        Y      Y          **
           **          RR    RR        0000          SS  SS       YY    YY          **
           **          RR     RR      00  00        SS             YY  YY           **
           **          RR    RR      00    00       SS              Y  Y            **
           **          RRRRRR        00    00        SSSS           YYYY            **
           **          RR    RR      00    00           SS          YY              **
           **          RR     RR      00  00          SS           YY              **
           **          RR     RR       0000         SS  SS         YY              **
           **          RR     RR         00          SSSS          YY              **
           **                                                                          **
           ******************************************************************************
           **  AUTOREN:  DIETMAR ALBRECHT                                              **
           **            DR. CLEMENS SCHMIDT                                           **
           ******************************************************************************
           **                                                                          **
           **    LEISTUNGSMERKMALE:                                                    **
           **    ------------------                                                    **
           **      - 10 PROZESSE                                                       **
           **      -  5 PRIORITAETEN                                                   **
           **      -  2 VERTAGUNGSMECHANISMEN                                          **
           **      -  ZEITVERWALTUNG: 'BELIEBIG' VIELE TIMER GLEICHZEITIG PRO PROZESS  **
           **      -  SCHEDULING-METHODE: DER MIT DER HOECHSTEN PRIORITAET AM LAENGSTEN **
           **                             WARTENDE PROZESS ERHAELT DEN PROZESSOR       **
           **                                                                          **
           **    LAUFZEITEN:                                                           **
           **    -----------                                                           **
           **      DIE LAUFZEITEN DER DEM ANWENDER ZUR VERFUEGUNG STEHENDEN PROZEDUREN **
           **      WURDEN AUF DER IN KAPITEL 5.2.4 BESCHRIEBENEN BAUGRUPPE ERMITTELT.  **
           **      SIE SIND IN DEN JEWEILIGEN PROZEDURKOEPFEN EINGETRAGEN.             **
           **      ALS MESSGERAET WURDE EIN SME SERIE-III BESTUECKT MIT DEM ENT-       **
           **      WICKLUNGSSYSTEM ICE88A BENUTZT. DIE ZEITEN WURDEN MIT DEM           **
           **      ICE-INTERNEN TIMER GEMESSEN.                                        **
           **      MESSBEDINGUNGEN: - 8088 CPU MIT 5 MHZ PROZESSORTAKT                 **
           **                       - SPEICHERZUGRIFF OHNE 'WAIT-STATES'               **
           **                                                                          **
           ******************************************************************************
           *****************************************************************************/

           $EJECT
```

```
1          ROSY: DO;

           /*****************************************************************
           **                        K O N S T A N T E N                    **
           *****************************************************************/

2   1      DECLARE FALSE          LITERALLY '0',
                   TRUE           LITERALLY '1',
                   FOREVER        LITERALLY 'WHILE TRUE',
                   NULL_SEGMENT   LITERALLY '0FFFFH',
                   NULL_OFFSET    LITERALLY '0FFFFH';

3   1      DECLARE PROCESS_MAX               LITERALLY    '10',
                   PRIORITAET_MAX            LITERALLY     '5',
                   TIME_SLOT_MAX             LITERALLY   '256',
                   SYSTEM_POOL_ELEMENT_MAX   LITERALLY   '128',
                   STACK_ARRAY_MAX           LITERALLY  '2560';

4   1      DECLARE NOT_PRESENT  LITERALLY '0',
                   PRESENT      LITERALLY '1';

5   1      DECLARE NICHT_INITIALISIERT  LITERALLY '0',
                   ABLAUFBEREIT         LITERALLY '1',
                   WARTEN_AUF_NACHRICHT LITERALLY '2',
                   WARTEN_AUF_INTERRUPT LITERALLY '3';

6   1      DECLARE TIMER_EVENT LITERALLY '0';

7   1      DECLARE IDLE_PID    LITERALLY '9';

8   1      DECLARE BP_DS_SIZE  LITERALLY '4';

           $EJECT
```

```
/*****************************************************************************
**                          S T R U K T U R E N                              **
*****************************************************************************/

 9  1    DECLARE QUEUE_HEADER_STRUCT  LITERALLY  'STRUCTURE
           (FIRST_IN_QUEUE  POINTER,
            LAST_IN_QUEUE   POINTER,
            NUM_IN_QUEUE    WORD)';

10  1    DECLARE PROCESS_LOAD_TABLE_STRUCT LITERALLY 'STRUCTURE
           (ZUSTAND     BYTE,
            PRIORITAET  BYTE,
            CS          SELECTOR,
            IP          WORD,
            SS          SELECTOR,
            SP          WORD)';

11  1    DECLARE PROCESS_CTRL_BLK_STRUCT LITERALLY 'STRUCTURE
           (FWD_LINK     POINTER,
            PROCESS_NR   BYTE,
            ZUSTAND      BYTE,
            SS           WORD,
            SP           WORD,
            PRIORITAET   BYTE,
            NEW_PRIOR    BYTE)';

12  1    DECLARE START_ADR_STRUCT LITERALLY 'STRUCTURE
           (IP  WORD,
            CS  SELECTOR)';

13  1    DECLARE GENERAL_POOL_ELEMENT_STRUCT LITERALLY  'STRUCTURE
           (FWD_LINK         POINTER,
            USER_DATA (23)   BYTE)';

14  1    DECLARE EVENT_STRUCT LITERALLY 'STRUCTURE
           (FWD_LINK     POINTER,
            EMPFAENGER   BYTE,
            ABSENDER     BYTE,
            EVENT_TYPE   BYTE,
            USER_DATA    (20) BYTE)';

15  1    DECLARE TIMER_STRUCT LITERALLY 'STRUCTURE
           (FWD_LINK_TQ  POINTER,
           BWD_LINK_TQ   POINTER,
           TIMER_TYPE    WORD,
           USER_CODE     WORD,
           PID           BYTE,
           TIME_SLOT     BYTE,
           BWD_LINK_TST POINTER,
           FWD_LINK_TST POINTER)';

16  1    DECLARE TIMER_MSG_STRUCT LITERALLY 'STRUCTURE
           (FWD_LINK     POINTER,
            EMPFAENGER   BYTE,
            ABSENDER     BYTE,
            EVENT_TYPE   BYTE,
            DUMMY        BYTE,
            TIMER_TYPE   WORD,
            USER_CODE    WORD)';

         $EJECT
```

```
/*****************************************************************************
**                          V A R I A B L E N                                **
*****************************************************************************/

17  1   DECLARE PROCESS_LOAD_TABLE (PROCESS_MAX) PROCESS_LOAD_TABLE_STRUCT
            DATA (    PRESENT, 4, 0FEABH, 00010H, 00050H, 01A08H,  /* REAL-TIME-PROCESS */
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                 NOT_PRESENT, 4, 00000H, 00000H, 00000H, 00000H,
                     PRESENT, 0, 00000H, 00000H, 00050H, 01D08H); /* IDLE-PROCESS */

18  1   DECLARE PROCESS_CTRL_BLK (PROCESS_MAX) PROCESS_CTRL_BLK_STRUCT;

19  1   DECLARE PROCESS_QUEUE (PRIORITAET_MAX) QUEUE_HEADER_STRUCT;

20  1   DECLARE CURRENT_PID BYTE;

21  1   DECLARE EVENT_QUEUE (PROCESS_MAX) QUEUE_HEADER_STRUCT,
            TIMER_QUEUE (PROCESS_MAX) QUEUE_HEADER_STRUCT PUBLIC;

22  1   DECLARE TIME_SLOT_TABLE (TIME_SLOT_MAX) QUEUE_HEADER_STRUCT PUBLIC,
            CURRENT_TIME_SLOT  BYTE  PUBLIC;

23  1   DECLARE SYSTEM_POOL  QUEUE_HEADER_STRUCT  PUBLIC,
            SYSTEM_POOL_ELEMENT (SYSTEM_POOL_ELEMENT_MAX)  GENERAL_POOL_ELEMENT_STRUCT;

24  1   DECLARE IDLE_LOOP_CTR  WORD;

25  1   DECLARE STACK_ARRAY (STACK_ARRAY_MAX)  BYTE;

26  1   DECLARE NULL_ADDRESS  POINTER;

27  1   DECLARE PRIORITAET       BYTE,        /* VARIABLEN FUER PROCESS_SCHEDULER */
            NEW_PROCESS_ADR  POINTER,
            NEW_PROCESS      BASED NEW_PROCESS_ADR  PROCESS_CTRL_BLK_STRUCT;

        $EJECT
```

```
/*****************************************************************************
**                    E X T E R N E   P R O Z E D U R E N                   **
*****************************************************************************/

1   HW_INIT: PROCEDURE EXTERNAL;
2   END HW_INIT;

1   INTERRUPT_INIT: PROCEDURE EXTERNAL;
2   END INTERRUPT_INIT;
    $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  TIMER_ASSIGN                                            **
**                                                                          **
**  FUNKTION     :  SETZEN EINES ZEITGLIEDES                                **
**                                                                          **
**  PARAMETER    :  TIME       : ANZAHL DER ZEITEINHEITEN                   **
**                  TIMER_TYPE : ANWENDERKENNZEICHEN DES ZEITGLIEDES        **
**                  USER_CODE  : ANWENDERKENNZEICHEN DES ZEITGLIEDES        **
**                                                                          **
**  LAUFZEIT     :  900 uS                                                  **
**                                                                          **
*****************************************************************************/

32   1    TIMER_ASSIGN: PROCEDURE (TIME, TIMER_TYPE, USER_CODE) REENTRANT PUBLIC;

33   2     DECLARE TIME       BYTE,
                 TIMER_TYPE WORD,
                 USER_CODE  WORD;

34   2     DECLARE TIME_SLOT  BYTE;
35   2     DECLARE TIMER_POOL_ELEMENT_ADR POINTER,
                 POOL_ELEMENT           BASED TIMER_POOL_ELEMENT_ADR  TIMER_STRUCT;


36   2     CALL GET_POOL (@SYSTEM_POOL, @TIMER_POOL_ELEMENT_ADR);
37   2     POOL_ELEMENT.TIMER_TYPE = TIMER_TYPE;
38   2     POOL_ELEMENT.USER_CODE  = USER_CODE;
39   2     POOL_ELEMENT.PID        = CURRENT_PID;
40   2     TIME_SLOT = (CURRENT_TIME_SLOT + TIME) MOD TIME_SLOT_MAX;
41   2     POOL_ELEMENT.TIME_SLOT  = TIME_SLOT;
42   2     DISABLE;
43   2     POOL_ELEMENT.BWD_LINK_TG  = TIMER_QUEUE(CURRENT_PID).LAST_IN_QUEUE;
44   2     CALL QUEUE (@TIMER_QUEUE(CURRENT_PID), TIMER_POOL_ELEMENT_ADR);
45   2     POOL_ELEMENT.BWD_LINK_TST = TIME_SLOT_TABLE (TIME_SLOT).LAST_IN_QUEUE;
46   2     CALL QUEUE (@TIME_SLOT_TABLE (TIME_SLOT),
                   @POOL_ELEMENT.FWD_LINK_TST);
47   2     ENABLE;


48   2    END TIMER_ASSIGN;

          $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  TIMER_RELEASE                                           **
**                                                                          **
**  FUNKTION     :  LOESCHEN EINES GESETZTEN ZEITGLIEDES                    **
**                                                                          **
**  PARAMETER    :  TIMER_TYPE : ANWENDERKENNZEICHEN DES ZEITGLIEDES        **
**                  USER_CODE  : ANWENDERKENNZEICHEN DES ZEITGLIEDES        **
**                                                                          **
**  LAUFZEIT     :  >830 uS                                                 **
**                                                                          **
*****************************************************************************/

49  1   TIMER_RELEASE: PROCEDURE (TIMER_TYPE, USER_CODE) REENTRANT PUBLIC;

50  2    DECLARE TIMER_TYPE WORD,
                 USER_CODE  WORD;

51  2    DECLARE TIMER_FOUND_FLAG  BYTE,
                 TIMER_ADR         POINTER,
                 TIMER             BASED TIMER_ADR TIMER_STRUCT,
                 PRE_TIMER_ADR     POINTER,
                 TIMER_MSG_ADR     POINTER,
                 TIMER_MSG         BASED TIMER_MSG_ADR TIMER_MSG_STRUCT,
                 NEXT_TIMER_ADR    POINTER,
                 NEXT_TIMER        BASED NEXT_TIMER_ADR  TIMER_STRUCT,
                 PRE_TIMER_MSG_ADR POINTER;


52  2    TIMER_FOUND_FLAG = FALSE;
53  2    TIMER_ADR        = TIMER_QUEUE(CURRENT_PID).FIRST_IN_QUEUE;
54  2    PRE_TIMER_ADR    = @TIMER_QUEUE(CURRENT_PID);
55  2    DISABLE;
56  2    DO WHILE (TIMER_FOUND_FLAG = FALSE)  AND
                (TIMER_ADR <> NULL_ADDRESS);
57  3       IF (TIMER.TIMER_TYPE = TIMER_TYPE)  AND
              (TIMER.USER_CODE  = USER_CODE)
58  3          THEN TIMER_FOUND_FLAG = TRUE;
59  4          ELSE DO; PRE_TIMER_ADR = TIMER_ADR;
61  4                   TIMER_ADR     = TIMER.FWD_LINK_TQ;
62  4               END;
63  3    END;
64  2    IF TIMER_FOUND_FLAG = TRUE
65  3       THEN DO; NEXT_TIMER_ADR = TIMER.FWD_LINK_TQ;
67  3                IF NEXT_TIMER_ADR <> NULL_ADDRESS
68  3                   THEN NEXT_TIMER.BWD_LINK_TQ = TIMER.BWD_LINK_TQ;
69  3                CALL DEQUEUE (@TIMER_QUEUE(CURRENT_PID), PRE_TIMER_ADR);
70  3                IF TIME_SLOT_TABLE(TIMER.TIME_SLOT).NUM_IN_QUEUE = 1
71  3                   THEN CALL DEQUEUE (@TIME_SLOT_TABLE(TIMER.TIME_SLOT),
                                            @TIME_SLOT_TABLE(TIMER.TIME_SLOT));
72  4                   ELSE DO; NEXT_TIMER_ADR = TIMER.FWD_LINK_TST;
74  4                            IF NEXT_TIMER_ADR <> NULL_ADDRESS
75  4                               THEN NEXT_TIMER.BWD_LINK_TST = TIMER.BWD_LINK_TST;
76  4                            PRE_TIMER_ADR = TIMER.BWD_LINK_TST;
77  4                            IF PRE_TIMER_ADR <> NULL_ADDRESS
78  4                               THEN CALL DEQUEUE (@TIME_SLOT_TABLE(TIMER.TIME_SLOT),
                                                       PRE_TIMER_ADR);
79  4                               ELSE CALL DEQUEUE (@TIME_SLOT_TABLE(TIMER.TIME_SLOT),
                                                       @TIME_SLOT_TABLE(TIMER.TIME_SLOT));
80  4                        END;
81  3                CALL RELEASE_POOL (@SYSTEM_POOL, TIMER_ADR);
82  3            END;

      $EJECT
```

```
                ELSE DO; TIMER_MSG_ADR     = EVENT_QUEUE(CURRENT_PID).FIRST_IN_QUEUE;
                         PRE_TIMER_MSG_ADR = @EVENT_QUEUE(CURRENT_PID);
                         DO WHILE (TIMER_FOUND_FLAG = FALSE) AND
                                  (TIMER_MSG_ADR <> NULL_ADDRESS);
                            IF (TIMER_MSG.EVENT_TYPE = TIMER_EVENT) AND
                               (TIMER_MSG.TIMER_TYPE = TIMER_TYPE)  AND
                               (TIMER_MSG.USER_CODE  = USER_CODE)
                               THEN TIMER_FOUND_FLAG = TRUE;
                               ELSE DO; PRE_TIMER_MSG_ADR = TIMER_MSG_ADR;
                                        TIMER_MSG_ADR     = TIMER_MSG.FWD_LINK;
                                    END;
                         END;
                         IF TIMER_FOUND_FLAG = TRUE
                            THEN DO; CALL DEQUEUE (@EVENT_QUEUE(CURRENT_PID), PRE_TIMER_MSG_ADR);
                                     CALL RELEASE_POOL (@SYSTEM_POOL, TIMER_MSG_ADR);
                                 END;
                            ELSE CALL ERROR;
                      END;

    END TIMER_RELEASE;

    $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  UNQUEUE                                                 **
**                                                                          **
**  FUNKTION     :  AUSKETTEN DES ERSTEN ELEMENTS EINER QUEUE.              **
**                  IST DIE QUEUE LEER, WIRD 'NULL' ZURUECKGEGEBEN.         **
**                                                                          **
**  PARAMETER    :  - QHDR_ADR    : ADRESSE DER QUEUE                       **
**                  - ZEIGER_ADR : ADRESSE EINES ZEIGERS AUF DEN DIE        **
**                                 ADRESSE DES AUSGEKETTETEN ELEMENTS       **
**                                 GESCHRIEBEN WIRD                         **
**                                                                          **
**  LAUFZEIT     :  168 uS - 197 uS                                         **
**                                                                          **
*****************************************************************************/

116   1     UNQUEUE: PROCEDURE (QHDR_ADR, ZEIGER_ADR) REENTRANT PUBLIC;

117   2      DECLARE QHDR_ADR      POINTER,
                     ZEIGER_ADR    POINTER;
118   2      DECLARE QHDR          BASED QHDR_ADR QUEUE_HEADER_STRUCT,
                     ZEIGER        BASED ZEIGER_ADR POINTER,
                     QUE_ELE_ADR   POINTER,
                     QUEUE_ELEMENT BASED QUE_ELE_ADR GENERAL_POOL_ELEMENT_STRUCT;


119   2      IF QHDR.NUM_IN_QUEUE = 0
120   2         THEN ZEIGER = NULL_ADDRESS;
121   3         ELSE DO; ZEIGER               = QHDR.FIRST_IN_QUEUE;
123   3                  QUE_ELE_ADR          = QHDR.FIRST_IN_QUEUE;
124   3                  QHDR.FIRST_IN_QUEUE  = QUEUE_ELEMENT.FWD_LINK;
125   3                  QHDR.NUM_IN_QUEUE    = QHDR.NUM_IN_QUEUE - 1;
126   3                  IF QHDR.NUM_IN_QUEUE = 0
127   3                     THEN QHDR.LAST_IN_QUEUE = NULL_ADDRESS;
128   3                  END;


129   2     END UNQUEUE;

            $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  ENQUEUE                                                 **
**                                                                          **
**  FUNKTION     :  EINKETTEN EINES NEUEN ELEMENTS AN EINE BELIEBIGE        **
**                  STELLE INNERHALB EINER QUEUE                            **
**                                                                          **
**  PARAMETER    :  - QHDR_ADR : ADRESSE DER QUEUE                          **
**                  - PRE_ADR  : ADRESSE DES QUEUE-ELEMENTS HINTER DEM DAS  **
**                               NEUE ELEMENT EINGEKETTET WERDEN SOLL.      **
**                               BEI EINKETTEN AN ERSTER STELLE:            **
**                               ADRESSE DER QUEUE                          **
**                  - NEW_ADR  : ADRESSE DES EINZUKETTENDEN ELEMENTS        **
**                                                                          **
**  LAUFZEIT     :  156 uS - 186 uS                                         **
**                                                                          **
*****************************************************************************/

130   1     ENQUEUE: PROCEDURE (QHDR_ADR, PRE_ADR, NEW_ADR) REENTRANT PUBLIC;

131   2      DECLARE QHDR_ADR  POINTER,
                     PRE_ADR   POINTER,
                     NEW_ADR   POINTER;
132   2      DECLARE QHDR         BASED QHDR_ADR  QUEUE_HEADER_STRUCT,
                     PRE_ELEMENT BASED PRE_ADR   GENERAL_POOL_ELEMENT_STRUCT,
                     NEW_ELEMENT BASED NEW_ADR   GENERAL_POOL_ELEMENT_STRUCT;


133   2      NEW_ELEMENT.FWD_LINK = PRE_ELEMENT.FWD_LINK;
134   2      PRE_ELEMENT.FWD_LINK = NEW_ADR;
135   2      QHDR.NUM_IN_QUEUE    = QHDR.NUM_IN_QUEUE + 1;
136   2      IF NEW_ELEMENT.FWD_LINK = NULL_ADDRESS
137   2         THEN QHDR.LAST_IN_QUEUE = NEW_ADR;


138   2     END ENQUEUE;

           $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  DEQUEUE                                                 **
**                                                                          **
**  FUNKTION     :  AUSKETTEN EINES QUEUE-ELEMENTS VON BELIEBIGER STELLE     **
**                  INNERHALB EINER QUEUE                                   **
**                                                                          **
**  PARAMETER    :  - QHDR_ADR : ADRESSE DER QUEUE                          **
**                  - PRE_ADR  : ADRESSE DES QUEUE-ELEMENTS, DAS IN DER     **
**                               QUEUE VOR DEM AUSZUKETTENDEN STEHT.        **
**                               BEIM AUSKETTEN DES ERSTEN QUEUE-ELEMENTS:  **
**                               ADRESSE DER QUEUE.                         **
**                                                                          **
**  LAUFZEIT     :  217 uS - 276 uS                                         **
**                                                                          **
*****************************************************************************/

139   1     DEQUEUE: PROCEDURE (QHDR_ADR, PRE_ADR) REENTRANT PUBLIC;

140   2      DECLARE QHDR_ADR  POINTER,
                     PRE_ADR   POINTER;
141   2      DECLARE QHDR           BASED QHDR_ADR  QUEUE_HEADER_STRUCT,
                     PRE_ELEMENT    BASED PRE_ADR   GENERAL_POOL_ELEMENT_STRUCT,
                     DELINK_ADR     POINTER,
                     DELINK_ELEMENT BASED DELINK_ADR GENERAL_POOL_ELEMENT_STRUCT;


142   2      IF PRE_ELEMENT.FWD_LINK <> NULL_ADDRESS
143   3         THEN DO; DELINK_ADR = PRE_ELEMENT.FWD_LINK;
145   3                  PRE_ELEMENT.FWD_LINK = DELINK_ELEMENT.FWD_LINK;
146   3                  IF PRE_ELEMENT.FWD_LINK = NULL_ADDRESS
147   3                     THEN QHDR.LAST_IN_QUEUE = PRE_ADR;
148   3                  QHDR.NUM_IN_QUEUE = QHDR.NUM_IN_QUEUE - 1;
149   3                  IF QHDR.NUM_IN_QUEUE = 0
150   3                     THEN QHDR.LAST_IN_QUEUE = NULL_ADDRESS;
151   3               END;


152   2     END DEQUEUE;

            $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  GET_POOL                                                **
**                                                                          **
**  FUNKTION     :  ANFORDERN EINES POOL-ELEMENTS. GET_POOL DARF NICHT       **
**                  IN INTERRUPT-PROZEDUREN VERWENDET WERDEN.               **
**                                                                          **
**  PARAMETER    :. - POOL_ADR     : ADRESSE DES POOLS VON DEM EIN ELEMENT   **
**                                   ANGEFORDERT WIRD.                      **
**                  - POOL_PTR_ADR : ADRESSE EINES ZEIGERS AUF DEN DIE       **
**                                   ADRESSE DES ZUR VERFUEGUNG GESTELLTEN  **
**                                   POOL-ELEMENTS GESCHRIEBEN WIRD.        **
**                                                                          **
**  LAUFZEIT     :  250 uS                                                  **
**                                                                          **
*****************************************************************************/

153  1   GET_POOL: PROCEDURE (POOL_ADR, POOL_PTR_ADR) REENTRANT PUBLIC;

154  2    DECLARE POOL_ADR     POINTER,
                 POOL_PTR_ADR POINTER;
155  2    DECLARE POOL         BASED POOL_ADR  QUEUE_HEADER_STRUCT;


156  2    DISABLE;
157  2    IF POOL.NUM_IN_QUEUE = 0
158  2       THEN CALL ERROR;
159  2    CALL UNQUEUE (POOL_ADR, POOL_PTR_ADR);
160  2    ENABLE;

161  2   END GET_POOL;

/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  GET_POOL_DIS                                            **
**                                                                          **
**  FUNKTION     :  ANFORDERN EINES POOL-ELEMENTS AUS INTERRUPT-PROZEDUREN   **
**                                                                          **
**  PARAMETER    :  - POOL_ADR     : ADRESSE DES POOLS VON DEM EIN ELEMENT   **
**                                   ANGEFORDERT WIRD.                      **
**                  - POOL_PTR_ADR : ADRESSE EINES ZEIGERS AUF DEN DIE       **
**                                   ADRESSE DES ZUR VERFUEGUNG GESTELLTEN  **
**                                   POOL-ELEMENTS GESCHRIEBEN WIRD.        **
**                                                                          **
**  LAUFZEIT     :  250 uS                                                  **
**                                                                          **
*****************************************************************************/

162  1   GET_POOL_DIS: PROCEDURE (POOL_ADR, POOL_PTR_ADR) REENTRANT PUBLIC;

163  2    DECLARE POOL_ADR     POINTER,
                 POOL_PTR_ADR POINTER;
164  2    DECLARE POOL         BASED POOL_ADR  QUEUE_HEADER_STRUCT;


165  2    DISABLE;
166  2    IF POOL.NUM_IN_QUEUE = 0
167  2       THEN CALL ERROR;
168  2    CALL UNQUEUE (POOL_ADR, POOL_PTR_ADR);

169  2   END GET_POOL_DIS;

    $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME : RELEASE_POOL                                             **
**                                                                          **
**  FUNKTION     : RUECKGABE EINES POOLELEMENTS. RELEASE_POOL DARF NICHT     **
**                 IN INTERRUPT-PROZEDUREN VERWENDET WERDEN.                **
**                                                                          **
**  PARAMETER    : - QHDR_ADR         : ADRESSE DES POOLS AN DEN DAS ELEMENT **
**                                      ZURUECKGEGEBEN WERDEN SOLL          **
**                 - POOL_ELEMENT_ADR : ADRESSE DES POOLELEMENTS           **
**                                                                          **
**  LAUFZEIT     : 245 uS                                                   **
**                                                                          **
*****************************************************************************/

170   1     RELEASE_POOL: PROCEDURE (QHDR_ADR, POOL_ELEMENT_ADR) REENTRANT PUBLIC;

171   2      DECLARE QHDR_ADR          POINTER,
                      POOL_ELEMENT_ADR  POINTER;


172   2      DISABLE;
173   2      CALL QUEUE (QHDR_ADR, POOL_ELEMENT_ADR);
174   2      ENABLE;

175   2     END RELEASE_POOL;

/*****************************************************************************
**                                                                          **
**  PROZEDURNAME : RELEASE_POOL_DIS                                         **
**                                                                          **
**  FUNKTION     : RUECKGABE EINES POOLELEMENTS AUS INTERRUPT-PROZEDUREN     **
**                                                                          **
**  PARAMETER    : - QHDR_ADR         : ADRESSE DES POOLS AN DEN DAS ELEMENT **
**                                      ZURUECKGEGEBEN WERDEN SOLL          **
**                 - POOL_ELEMENT_ADR : ADRESSE DES POOLELEMENTS           **
**                                                                          **
**  LAUFZEIT     : 245 uS                                                   **
**                                                                          **
*****************************************************************************/

176   1     RELEASE_POOL_DIS: PROCEDURE (QHDR_ADR, POOL_ELEMENT_ADR) REENTRANT PUBLIC;

177   2      DECLARE QHDR_ADR          POINTER,
                      POOL_ELEMENT_ADR  POINTER;


178   2      DISABLE;
179   2      CALL QUEUE (QHDR_ADR, POOL_ELEMENT_ADR);

180   2     END RELEASE_POOL_DIS;

      $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  UNSUSPEND                                               **
**                                                                          **
**  FUNKTION     :  PROZESS IN PROZESS-QUEUE EINKETTEN UND DEN PROCESS-      **
**                  SCHEDULER AUFRUFEN FALLS SEINE PRIORITAET HOEHER IST    **
**                  ALS DIE DES GERADE LAUFENDEN PROZESSES                  **
**                                                                          **
**  PARAMETER    :  - PID      : PROZESS-NUMMER                            **
**                  - ZUSTAND  : ALTER ZUSTAND DES PROZESSES               **
**                                                                          **
*****************************************************************************/

181  1   UNSUSPEND: PROCEDURE (PID, ZUSTAND) REENTRANT;

182  2    DECLARE PID     BYTE,
             ZUSTAND BYTE;

183  2    IF PROCESS_CTRL_BLK(PID).ZUSTAND = ZUSTAND
184  3       THEN DO; PROCESS_CTRL_BLK(PID).ZUSTAND = ABLAUFBEREIT;
186  3                CALL QUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(PID).PRIORITAET),
                                 @PROCESS_CTRL_BLK(PID));
187  3                IF PROCESS_CTRL_BLK(PID).PRIORITAET > PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET
188  3                   THEN CALL PROCESS_SCHEDULER;
189  3             END;

190  2   END UNSUSPEND;

/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  SUSPEND                                                 **
**                                                                          **
**  FUNKTION     :  VERTAGEN EINES PROZESSES                                **
**                                                                          **
**  PARAMETER    :  - ZUSTAND : NEUER ZUSTAND DES PROZESSES                 **
**                                                                          **
*****************************************************************************/

191  1   SUSPEND: PROCEDURE (ZUSTAND) REENTRANT;

192  2    DECLARE ZUSTAND BYTE;

193  2    DECLARE PTR     POINTER;

194  2    CALL UNQUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET), @PTR);
195  2    PROCESS_CTRL_BLK(CURRENT_PID).ZUSTAND = ZUSTAND;
196  2    CALL PROCESS_SCHEDULER;

197  2   END SUSPEND;

     $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  PROCESS_SCHEDULER                                       **
**                                                                          **
**  FUNKTION     :  PROZESSWECHSEL DURCHFUEHREN                             **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

198   1     PROCESS_SCHEDULER: PROCEDURE REENTRANT PUBLIC;


199   2      PROCESS_CTRL_BLK(CURRENT_PID).SS = STACKBASE;
200   2      PROCESS_CTRL_BLK(CURRENT_PID).SP = STACKPTR;
201   2      PRIORITAET = PRIORITAET_MAX - 1;
202   2      DO WHILE PROCESS_QUEUE(PRIORITAET).NUM_IN_QUEUE = 0;
203   3         PRIORITAET = PRIORITAET - 1;
204   3      END;
205   2      NEW_PROCESS_ADR = PROCESS_QUEUE(PRIORITAET).FIRST_IN_QUEUE;
206   2      CURRENT_PID     = NEW_PROCESS.PROCESS_NR;
207   2      STACKBASE       = PROCESS_CTRL_BLK(CURRENT_PID).SS;
208.  2      STACKPTR        = PROCESS_CTRL_BLK(CURRENT_PID).SP;


209   2     END PROCESS_SCHEDULER;

           $EJECT
```

```
/*****************************************************************
**                                                              **
**  PROZEDURNAME :  SEND_EVENT                                  **
**                                                              **
**  FUNKTION     :  VERSENDEN EINER NACHRICHT AN EINEN ANDEREN PROZESS **
**                                                              **
**  PARAMETER    :  - REC_PID   : PROZESS-NUMMER DES EMPFANGSPROZESSES **
**                  - EVENT_PTR : ADRESSE DES POOLELEMENTS, DAS DIE **
**                                NACHRICHT ENTHAELT            **
**                                                              **
**  LAUFZEIT     :   286 uS  WENN ZIELPROZESS ABLAUFBEREIT      **
**                   595 uS  WENN ZIELPROZESS VERTAGT OHNE PROZESSWECHSEL **
**                  >760 uS  WENN ZIELPROZESS VERTAGT MIT PROZESSWECHSEL **
**                                                              **
*****************************************************************/

210  1   SEND_EVENT: PROCEDURE (REC_PID, EVENT_PTR) REENTRANT PUBLIC;

211  2    DECLARE REC_PID   BYTE,
             EVENT_PTR POINTER;

212  2    DECLARE EVENT BASED EVENT_PTR EVENT_STRUCT;


213  2    EVENT.ABSENDER   = CURRENT_PID;
214  2    EVENT.EMPFAENGER = REC_PID;
215  2    DISABLE;
216  2    CALL QUEUE (@EVENT_QUEUE(REC_PID), EVENT_PTR);
217  2    IF PROCESS_CTRL_BLK(REC_PID).ZUSTAND <> ABLAUFBEREIT
218  2       THEN CALL UNSUSPEND (REC_PID, WARTEN_AUF_NACHRICHT);
219  2    ENABLE;


220  2   END SEND_EVENT;

        $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  RECEIVE_EVENT                                           **
**                                                                          **
**  FUNKTION     :  EMPFANGEN EINER NACHRICHT AUS DER EVENT-QUEUE            **
**                                                                          **
**  PARAMETER    :  EVENT_PTR_ADR : ADRESSE EINES ZEIGERS AUF DEN DIE        **
**                                  ADRESSE DES POOLELEMENTS GESCHRIEBEN    **
**                                  WIRD, DAS DIE NACHRICHT ENTHAELT        **
**                                                                          **
**  LAUFZEIT     :   250 uS WENN EVENT VORHANDEN                            **
**                  >520 uS BEI PROZESSWECHSEL                              **
**                                                                          **
*****************************************************************************/

221   1     RECEIVE_EVENT: PROCEDURE (EVENT_PTR_ADR) REENTRANT PUBLIC;

222   2      DECLARE EVENT_PTR_ADR POINTER;

223   2      DECLARE EVENT_PTR BASED EVENT_PTR_ADR POINTER;


224   2      DISABLE;
225   2      IF EVENT_QUEUE(CURRENT_PID).NUM_IN_QUEUE = 0
226   2         THEN CALL SUSPEND (WARTEN_AUF_NACHRICHT);
227   2      CALL UNQUEUE (@EVENT_QUEUE(CURRENT_PID), EVENT_PTR_ADR);
228   2      ENABLE;


229   2     END RECEIVE_EVENT;

        $EJECT
```

```
                /*****************************************************************
                **                                                              **
                **  PROZEDURNAME :  WAIT_INTERRUPT                              **
                **                                                              **
                **  FUNKTION     :  VERTAGEN MIT ZUSTAND 'WARTEN_AUF_INTERRUPT' **
                **                                                              **
                **  PARAMETER    :  KEINE                                       **
                **                                                              **
                **  LAUFZEIT     :  >560 uS                                     **
                **                                                              **
                *****************************************************************/

230   1         WAIT_INTERRUPT: PROCEDURE REENTRANT PUBLIC;

231   2          DISABLE;
232   2          CALL SUSPEND (WARTEN_AUF_INTERRUPT);
233   2          ENABLE;

234   2         END WAIT_INTERRUPT;


                /*****************************************************************
                **                                                              **
                **  PROZEDURNAME :  SIGNAL_INTERRUPT                            **
                **                                                              **
                **  FUNKTION     :  PROZESS ABLAUFBEREIT MACHEN, FALLS ER SICH IM **
                **                  ZUSTAND 'WARTEN_AUF_INTERRUPT' BEFINDET     **
                **                                                              **
                **  PARAMETER    :  PID : PROZESS-NUMMER                        **
                **                                                              **
                **  LAUFZEIT     :  >508 uS MIT PROZESSWECHSEL                  **
                **                                                              **
                *****************************************************************/


235   1         SIGNAL_INTERRUPT: PROCEDURE (PID) REENTRANT PUBLIC;

236   2          DECLARE PID BYTE;

237   2          CALL UNSUSPEND (PID, WARTEN_AUF_INTERRUPT);

238   2         END SIGNAL_INTERRUPT;

                $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  SET_PRIORITY                                            **
**                                                                          **
**  FUNKTION     :  PROZESS EINE ANDERE PRIORITAET ZUWEISEN                  **
**                                                                          **
**  PARAMETER    :  NEW_PRIOR : NEUE PRIORITAET                             **
**                                                                          **
**  LAUFZEIT     :  627 uS                                                  **
**                                                                          **
*****************************************************************************/

239   1     SET_PRIORITY: PROCEDURE (NEW_PRIOR) REENTRANT PUBLIC;

240   2      DECLARE NEW_PRIOR BYTE;

241   2      DISABLE;
242   2      PROCESS_CTRL_BLK(CURRENT_PID).NEW_PRIOR = NEW_PRIOR;
243   2      CALL DEQUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET),
                           @PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET));
244   2      CALL ENQUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).NEW_PRIOR),
                           @PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).NEW_PRIOR),
                           @PROCESS_CTRL_BLK(CURRENT_PID));
245   2      ENABLE;

246   2     END SET_PRIORITY;

/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  RESET_PRIORITY                                          **
**                                                                          **
**  FUNKTION     :  PROZESS AUF SEINE URSPRUENGLICHE PRIORITAET SETZEN       **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
**  LAUFZEIT     :  560 uS                                                  **
**                                                                          **
*****************************************************************************/

247   1     RESET_PRIORITY: PROCEDURE REENTRANT PUBLIC;

248   2      DISABLE;
249   2      CALL DEQUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).NEW_PRIOR),
                           @PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).NEW_PRIOR));
250   2      CALL ENQUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET),
                           @PROCESS_QUEUE(PROCESS_CTRL_BLK(CURRENT_PID).PRIORITAET),
                           @PROCESS_CTRL_BLK(CURRENT_PID));
251   2      ENABLE;

252   2     END RESET_PRIORITY;

      $EJECT
```

```
                /*****************************************************************
                **                                                              **
                **   PROZEDURNAME :  INITIALIZE_BS                              **
                **                                                              **
                **   FUNKTION     :  INITIALISIERUNG DER BETRIEBSSYSTEM-DATEN    **
                **                                                              **
                **   PARAMETER    :  KEINE                                      **
                **                                                              **
                *****************************************************************/

253   1       INITIALIZE_BS: PROCEDURE REENTRANT;

254   2        DECLARE VAR BYTE;


255   2        DO VAR = 0 TO PROCESS_MAX - 1;
256   3           TIMER_QUEUE(VAR).FIRST_IN_QUEUE = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
257   3           TIMER_QUEUE(VAR).LAST_IN_QUEUE  = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
258   3           TIMER_QUEUE(VAR).NUM_IN_QUEUE   = 0;

259   3           EVENT_QUEUE(VAR).FIRST_IN_QUEUE = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
260   3           EVENT_QUEUE(VAR).LAST_IN_QUEUE  = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
261   3           EVENT_QUEUE(VAR).NUM_IN_QUEUE   = 0;
262   3        END;

263   2        DO VAR = 0 TO PRIORITAET_MAX - 1;
264   3           PROCESS_QUEUE(VAR).FIRST_IN_QUEUE = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
265   3           PROCESS_QUEUE(VAR).LAST_IN_QUEUE  = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
266   3           PROCESS_QUEUE(VAR).NUM_IN_QUEUE   = 0;
267   3        END;

268   2           SYSTEM_POOL.FIRST_IN_QUEUE = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
269   2           SYSTEM_POOL.LAST_IN_QUEUE  = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
270   2           SYSTEM_POOL.NUM_IN_QUEUE   = 0;

271   2        DO VAR = 0 TO TIME_SLOT_MAX - 1;
272   3           TIME_SLOT_TABLE(VAR).FIRST_IN_QUEUE = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
273   3           TIME_SLOT_TABLE(VAR).LAST_IN_QUEUE  = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
274   3           TIME_SLOT_TABLE(VAR).NUM_IN_QUEUE   = 0;
275   3        END;

276   2        DO VAR = 0 TO SYSTEM_POOL_ELEMENT_MAX - 1;
277   3           CALL QUEUE (@SYSTEM_POOL, @SYSTEM_POOL_ELEMENT(VAR));
278   3        END;

279   2        NULL_ADDRESS = BUILD$PTR (NULL_SEGMENT, NULL_OFFSET);
280   2        CURRENT_PID = IDLE_PID;


281   2       END INITIALIZE_BS;

               $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  START_PROCESSES                                         **
**                                                                          **
**  FUNKTION     :  PROZESS FUER DEN ERSTEN START VORBEREITEN                **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

START_PROCESSES: PROCEDURE REENTRANT;

 DECLARE PID BYTE;
 DECLARE STACK_ADR  POINTER,
         STACK      BASED STACK_ADR  START_ADR_STRUCT;
 DECLARE PCB_SS_ADR POINTER,
         PCB_SS     BASED PCB_SS_ADR SELECTOR;


 DO PID = 0 TO PROCESS_MAX - 1;
    IF PROCESS_LOAD_TABLE(PID).ZUSTAND = PRESENT
       THEN DO; PROCESS_CTRL_BLK(PID).PROCESS_NR = PID;
                PROCESS_CTRL_BLK(PID).ZUSTAND    = ABLAUFBEREIT;
                PCB_SS_ADR                       = @PROCESS_CTRL_BLK(PID).SS;
                PCB_SS                           = PROCESS_LOAD_TABLE(PID).SS;
                PROCESS_CTRL_BLK(PID).SP         = PROCESS_LOAD_TABLE(PID).SP - SIZE (STACK) - BP_DS_SIZE;
                PROCESS_CTRL_BLK(PID).PRIORITAET = PROCESS_LOAD_TABLE(PID).PRIORITAET;
                PROCESS_CTRL_BLK(PID).NEW_PRIOR  = PROCESS_LOAD_TABLE(PID).PRIORITAET;

                STACK_ADR = BUILD$PTR (PCB_SS, PROCESS_CTRL_BLK(PID).SP + BP_DS_SIZE);
                STACK.CS  = PROCESS_LOAD_TABLE(PID).CS;
                STACK.IP  = PROCESS_LOAD_TABLE(PID).IP;

                CALL QUEUE (@PROCESS_QUEUE(PROCESS_CTRL_BLK(PID).PRIORITAET),
                            @PROCESS_CTRL_BLK(PID));
            END;
       ELSE PROCESS_CTRL_BLK(PID).ZUSTAND = NICHT_INITIALISIERT;
 END;

END START_PROCESSES;

$EJECT
```

```
                /*****************************************************************
                **                                                              **
                **  PROZEDURNAME :  ERROR                                       **
                **                                                              **
                **  FUNKTION     :  HALT                                        **
                **                                                              **
                **  PARAMETER    :  KEINE                                       **
                **                                                              **
                *****************************************************************/

304   1         ERROR: PROCEDURE REENTRANT;

305   2          DISABLE;
306   2          HALT;

307   2         END ERROR;

                $EJECT
```

```
/*****************************************************************************
**                                                                          **
**              B E T R I E B S S Y S T E M - S T A R T                     **
**                                                                          **
*****************************************************************************/

306   1     DISABLE;
309   1     CALL HW_INIT;
310   1     CALL INTERRUPT_INIT;
311   1     CALL INITIALIZE_BS;
312   1     CALL START_PROCESSES;
313   1     CALL PROCESS_SCHEDULER;

314   1     ENABLE;

/*****************************************************************************
**                                                                          **
**                          I D L E _ P R O C E S S                         **
**                                                                          **
*****************************************************************************/

315   1     IDLE_PROCESS: DO FOREVER;
316   2      IDLE_LOOP_CTR = IDLE_LOOP_CTR + 1;
317   2     END IDLE_PROCESS;


318   1     END ROSY;
```

```
MODULE INFORMATION:

     CODE AREA SIZE      = 0AB1H   2737D
     CONSTANT AREA SIZE  = 0000H      0D
     VARIABLE AREA SIZE  = 2309H   8969D
     MAXIMUM STACK SIZE  = 002AH     42D
     1064 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
SE..E... III PL/M-86 V2.0 COMPILATION OF MODULE REAL_TIME_MODUL
OBJECT MODULE PLACED IN :F5:RTP.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:RTP.PLM


                $DEBUG
                $LARGE
                $OPTIMIZE(2)
                $PAGELENGTH (70)
                $PAGEWIDTH(131)


  1             REAL_TIME_MODUL: DO;

                /*****************************************************************
                **                          L I T E R A L S                     **
                *****************************************************************/

  2    1        DECLARE FALSE    LITERALLY '0',
                        TRUE     LITERALLY '1',
                        FOREVER LITERALLY 'WHILE 0 = 0';

  3    1        DECLARE TIME_SLOT_MAX           LITERALLY '256',
                        SYSTEM_POOL_ELEMENT_MAX LITERALLY '128',
                        TIMER_LINK_DIFF         LITERALLY  '18',
                        PROCESS_MAX             LITERALLY  '10';

  4    1        DECLARE TIMER_EVENT LITERALLY '0';

                /*****************************************************************
                **              B E T R I E B S S Y S T E M - S T R U K T U R E N  **
                *****************************************************************/

  5    1        DECLARE QUEUE_HEADER_STRUCT  LITERALLY
                        'STRUCTURE (FIRST_IN_QUEUE  POINTER,
                                    LAST_IN_QUEUE   POINTER,
                                    NUM_IN_QUEUE    WORD)';

  6    1        DECLARE TIMER_STRUCT LITERALLY 'STRUCTURE
                        (FWD_LINK_TQ  POINTER,
                        BWD_LINK_TQ  POINTER,
                        TIMER_TYPE   WORD,
                        USER_CODE    WORD,
                        PID          BYTE,
                        TIME_SLOT    BYTE,
                        BWD_LINK_TST POINTER,
                        FWD_LINK_TST POINTER)';

  7    1        DECLARE TIMER_MSG_STRUCT LITERALLY 'STRUCTURE
                        (FWD_LINK    POINTER,
                         EMPFAENGER  BYTE,
                         ABSENDER    BYTE,
                         EVENT_TYPE  BYTE,
                         DUMMY       BYTE,
                         TIMER_TYPE  WORD,
                         USER_CODE   WORD)';

  8    1        DECLARE TIME_SLOT_TABLE (TIME_SLOT_MAX) QUEUE_HEADER_STRUCT EXTERNAL;
  9    1        DECLARE TIMER_QUEUE (PROCESS_MAX)       QUEUE_HEADER_STRUCT EXTERNAL;
 10    1        DECLARE SYSTEM_POOL                     QUEUE_HEADER_STRUCT EXTERNAL;
 11    1        DECLARE CURRENT_TIME_SLOT               BYTE                EXTERNAL;

                $EJECT
```

```
                /*****************************************************************
                **          B E T R I E B S S Y S T E M - P R O Z E D U R E N    **
                *****************************************************************/

12   1          SEND_EVENT: PROCEDURE (REC_PID, EVENT_PTR) EXTERNAL;

13   2           DECLARE REC_PID   BYTE,
                         EVENT_PTR POINTER;

14   2          END SEND_EVENT;

15   1          WAIT_INTERRUPT: PROCEDURE EXTERNAL;

16   2          END WAIT_INTERRUPT;

17   1          UNQUEUE: PROCEDURE (QHDR_ADR, ZEIGER_ADR) EXTERNAL;

18   2           DECLARE QHDR_ADR      POINTER,
                         ZEIGER_ADR    POINTER;

19   2          END UNQUEUE;

20   1          DEQUEUE: PROCEDURE (QHDR_ADR, PRE_ADR) EXTERNAL;

21   2           DECLARE QHDR_ADR  POINTER,
                         PRE_ADR   POINTER;

22   2          END DEQUEUE;

                $EJECT
```

```
                /*****************************************************************
                **                                                              **
                **  PROZEDURNAME :  REAL_TIME_PROCESS                           **
                **                                                              **
                **  FUNKTION     :  ABGELAUFENE ZEITAUFTRAEGE BEARBEITEN        **
                **                                                              **
                **  PARAMETER    :  KEINE                                       **
                **                                                              **
                *****************************************************************/

 23   1       REAL_TIME_PROCESS: PROCEDURE REENTRANT;

 24   2        DECLARE TIMER_PTR POINTER,
                      TIMER     BASED TIMER_PTR  TIMER_STRUCT,
                      TIMER_ADR STRUCTURE (OFF WORD, SEG WORD) AT (@TIMER_PTR);
 25   2        DECLARE SYSTEM_POOL_PTR     POINTER,
                      SYSTEM_POOL_ELEMENT BASED SYSTEM_POOL_PTR  TIMER_MSG_STRUCT;


 26   2        ENABLE;
 27   2        DO FOREVER;
 28   3           CALL WAIT_INTERRUPT;
 29   3           CURRENT_TIME_SLOT = CURRENT_TIME_SLOT + 1;
 30   3           DO WHILE TIME_SLOT_TABLE(CURRENT_TIME_SLOT).NUM_IN_QUEUE <> 0;
 31   4              CALL UNQUEUE (@TIME_SLOT_TABLE(CURRENT_TIME_SLOT), @TIMER_PTR);
 32   4              TIMER_ADR.OFF = TIMER_ADR.OFF - TIMER_LINK_DIFF;
 33   4              IF TIMER_QUEUE(TIMER.PID).FIRST_IN_QUEUE = TIMER_PTR
 34   4                 THEN CALL DEQUEUE (@TIMER_QUEUE(TIMER.PID), @TIMER_QUEUE(TIMER.PID));
 35   4                 ELSE CALL DEQUEUE (@TIMER_QUEUE(TIMER.PID), TIMER.BWD_LINK_TQ);
 36   4              SYSTEM_POOL_PTR                 = TIMER_PTR;
 37   4              SYSTEM_POOL_ELEMENT.EVENT_TYPE = TIMER_EVENT;
 38   4              CALL SEND_EVENT (TIMER.PID, SYSTEM_POOL_PTR);
 39   4           END;
 40   3           ENABLE;
 41   3        END;


 42   2       END REAL_TIME_PROCESS;


 43   1       END REAL_TIME_MODUL;



MODULE INFORMATION:

     CODE AREA SIZE     = 00E6H    230D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0018H     24D
     148 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Anhang 2:

Compilerlisting des Anwenderprozesses »Überwachung«

```
SERIES-III PL/M-86 V2.0 COMPILATION OF MODULE UEBERWACHUNG
OBJECT MODULE PLACED IN :F5:UEBW.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:UEBW.PLM


            $DEBUG
            $OPTIMIZE (2)
            $PAGELENGTH (70)
            $PAGEWIDTH (131)



            UEBERWACHUNG: DO;


            /*************************************************************************
            **                                                                      **
            **           B E T R I E B S S Y S T E M - V A R I A B L E              **
            **                                                                      **
            *************************************************************************/

 2   1      DECLARE SYSTEM_POOL (*) BYTE EXTERNAL;


            /*************************************************************************
            **                                                                      **
            **           B E T R I E B S S Y S T E M - S T R U K T U R E N          **
            **                                                                      **
            *************************************************************************/

 3   1      DECLARE EVENT_STRUCT LITERALLY 'STRUCTURE
                    (FWD_LINK    POINTER,
                     EMPFAENGER  BYTE,
                     ABSENDER    BYTE,
                     EVENT_TYPE  BYTE,
                     JOBCODE     BYTE,
                     PARAMETER1  BYTE)';

 4   1      DECLARE TIMER_MSG_STRUCT LITERALLY 'STRUCTURE
                    (FWD_LINK    POINTER,
                    EMPFAENGER  BYTE,
                    ABSENDER    BYTE,
                    EVENT_TYPE  BYTE,
                    DUMMY       BYTE,
                    TIMER_TYPE  WORD,
                    USER_CODE   WORD)';

            $EJECT
```

```
/*****************************************************************
**                                                              **
**          B E T R I E B S S Y S T E M - P R O Z E D U R E N   **
**                                                              **
*****************************************************************/

5   1   RELEASE_POOL: PROCEDURE (QHDR_ADR, POOL_ELEMENT_ADR) EXTERNAL;

6   2    DECLARE QHDR_ADR          POINTER,
             POOL_ELEMENT_ADR  POINTER;

7   2   END RELEASE_POOL;

8   1   SEND_EVENT: PROCEDURE (REC_PID, EVENT_PTR) EXTERNAL;

9   2    DECLARE REC_PID    BYTE,
             EVENT_PTR  POINTER;

10  2   END SEND_EVENT;

11  1   RECEIVE_EVENT: PROCEDURE (EVENT_PTR_ADR) EXTERNAL;

12  2    DECLARE EVENT_PTR_ADR POINTER;

13  2   END RECEIVE_EVENT;

14  1   TIMER_ASSIGN: PROCEDURE (TIME, TIMER_TYPE, USER_CODE) EXTERNAL;

15  2    DECLARE TIME        BYTE,
             TIMER_TYPE  WORD,
             USER_CODE   WORD;

16  2   END TIMER_ASSIGN;

17  1   TIMER_RELEASE: PROCEDURE (TIMER_TYPE, USER_CODE) EXTERNAL;

18  2    DECLARE TIMER_TYPE  WORD,
             USER_CODE   WORD;

19  2   END TIMER_RELEASE;

        $EJECT
```

```
/*****************************************************************************
**                                                                          **
**                E X T E R N E   P R O Z E D U R E N                       **
**                                                                          **
*****************************************************************************/

1     UEB_PROCESS_INITIALIZATION: PROCEDURE EXTERNAL;
2     END UEB_PROCESS_INITIALIZATION;

1     EXCEPTION_ROUTINE: PROCEDURE EXTERNAL;
2     END EXCEPTION_ROUTINE;

1     STARTKONTAKTE_BEWERTEN: PROCEDURE EXTERNAL;
2     END STARTKONTAKTE_BEWERTEN;

1     STARTKONTAKTE_SPERREN: PROCEDURE EXTERNAL;
2     END STARTKONTAKTE_SPERREN;

1     STATISTIK: PROCEDURE EXTERNAL;
2     END STATISTIK;

1     STATISTIK_AUSGEBEN: PROCEDURE EXTERNAL;
2     END STATISTIK_AUSGEBEN;

1     FEHLSTART_VERURSACHER: PROCEDURE WORD EXTERNAL;
2     END FEHLSTART_VERURSACHER;
      $EJECT
```

```
          /*****************************************************************************
          **                                                                          **
          **                    K O N S T A N T E N                                   **
          **                                                                          **
          *****************************************************************************/

34   1    DECLARE FOREVER LITERALLY 'WHILE 0 = 0';

          /** EVENT_TYPE **/

35   1    DECLARE TIMER     LITERALLY '0',
                 UM_UEB    LITERALLY '1',
                 BED_UEB   LITERALLY '2',
                 UEB_INDIV LITERALLY '3';

          /** UM_UEB_JOBCODES **/

36   1    DECLARE FEHLSTART LITERALLY '0',
                 GUTSTART  LITERALLY '1';

          /** BED_UEB_JOBCODES **/

37   1    DECLARE FREIGABE LITERALLY '0',
                 STOP     LITERALLY '1';

          /** UEB_INDIV_JOBCODES **/

38   1    DECLARE UEB_FEHLSTART LITERALLY '0';

          /** UEB_STATES **/

39   1    DECLARE RUHE  LITERALLY '0',
                 WARTE LITERALLY '1',
                 FREI  LITERALLY '2';

          /** TIMER_CODES **/

40   1    DECLARE T_STATISTIK LITERALLY '0',
                 NOT_USED    LITERALLY '0FFH';

          /** ZEITEN **/

41   1    DECLARE STATISTIK_ZEIT  LITERALLY '50';

          /** PROZESS_NUMMERN **/

42   1    DECLARE INDIV_RENNEN LITERALLY '10H';

          $EJECT
```

```
          /*****************************************************************************
          **                                                                          **
          **                        V A R I A B L E N                                 **
          **                                                                          **
          *****************************************************************************/

43    1   DECLARE EVENT_POINTER  POINTER,
                  EVENT          BASED EVENT_POINTER  EVENT_STRUCT,
                  TIMER_EVENT    BASED EVENT_POINTER  TIMER_MSG_STRUCT;

44    1   DECLARE UEB_STATES     BYTE;

          $EJECT
```

```
/*****************************************************************************
**                                                                           **
**  PROZEDURNAME :  UEBERWACHUNG_PROZESS                                     **
**                                                                           **
**  FUNKTION     :  PROZESSRAHMEN                                            **
**                                                                           **
**  PARAMETER    :  KEINE                                                    **
**                                                                           **
*****************************************************************************/

1      UEBERWACHUNG_PROZESS: PROCEDURE PUBLIC;

2       CALL UEB_PROCESS_INITIALIZATION;

2       DO FOREVER;

3          CALL RECEIVE_EVENT (@EVENT_POINTER);

3          IF EVENT.EVENT_TYPE <= BED_UEB
3             THEN DO CASE EVENT.EVENT_TYPE;

        /** TIMER **/
5                     DO; IF UEB_STATES = FREI  AND  TIMER_EVENT.TIMER_TYPE = T_STATISTIK
5                            THEN CALL PROC_E;
5                            ELSE CALL EXCEPTION_ROUTINE;
5                     END;

        /** UM_UEB **/
5                     DO; IF UEB_STATES = WARTE AND EVENT.JOBCODE = FEHLSTART
5                            THEN CALL PROC_C;
5                            ELSE IF UEB_STATES = WARTE AND EVENT.JOBCODE = GUTSTART
5                                    THEN CALL PROC_D;
5                                    ELSE CALL EXCEPTION_ROUTINE;
5                     END;

        /** BED_UEB **/
5                     DO; IF UEB_STATES = RUHE AND EVENT.JOBCODE = FREIGABE
5                            THEN CALL PROC_A;
5                            ELSE IF UEB_STATES = FREI AND EVENT.JOBCODE = STOP
5                                    THEN CALL PROC_B;
5                                    ELSE CALL EXCEPTION_ROUTINE;
5                     END;

4                  END;
3              ELSE CALL RELEASE_POOL (@SYSTEM_POOL, EVENT_POINTER);
3       END;

2       END UEBERWACHUNG_PROZESS;

        $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  PROC_A                                                  **
**                                                                          **
**  FUNKTION     :  BEARBEITUNG DES JOBCODES 'FREIGABE' IM ZUSTAND 'RUHE'   **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

74   1    PROC_A: PROCEDURE;

75   2     CALL STARTKONTAKTE_BEWERTEN;
76   2     UEB_STATES = WARTE;
77   2     CALL RELEASE_POOL (@SYSTEM_POOL, EVENT_POINTER);

78   2    END PROC_A;
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  PROC_B                                                  **
**                                                                          **
**  FUNKTION     :  BEARBEITUNG DES JOBCODES 'STOP' IM ZUSTAND 'FREI'       **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

79   1    PROC_B: PROCEDURE;

80   2     CALL STATISTIK;
81   2     CALL TIMER_RELEASE (T_STATISTIK, NOT_USED);
82   2     UEB_STATES = RUHE;
83   2     CALL RELEASE_POOL (@SYSTEM_POOL, EVENT_POINTER);

84   2    END PROC_B;
     $EJECT
```

```
/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  PROC_C                                                  **
**                                                                          **
**  FUNKTION     :  BEARBEITUNG DES JOBCODES 'FEHLSTART' IM ZUSTAND 'WARTE'  **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

85   1      PROC_C: PROCEDURE;

86   2       DECLARE SUENDER WORD;

87   2       CALL STARTKONTAKTE_SPERREN;

88   2       SUENDER = FEHLSTART_VERURSACHER;

89   2       EVENT.EVENT_TYPE = UEB_INDIV;
90   2       EVENT.JOBCODE    = UEB_FEHLSTART;
91   2       EVENT.PARAMETER1 = SUENDER;

92   2       CALL SEND_EVENT (INDIV_RENNEN, EVENT_POINTER);

93   2       UEB_STATES = RUHE;

94   2      END PROC_C;



/*****************************************************************************
**                                                                          **
**  PROZEDURNAME :  PROC_D                                                  **
**                                                                          **
**  FUNKTION     :  BEARBEITUNG DES JOBCODES  GUTSTART' IM ZUSTAND 'WARTE'   **
**                                                                          **
**  PARAMETER    :  KEINE                                                   **
**                                                                          **
*****************************************************************************/

95   1      PROC_D: PROCEDURE;

96   2       CALL TIMER_ASSIGN (STATISTIK_ZEIT, T_STATISTIK, NOT_USED);

97   2       UEB_STATES = FREI;

98   2       CALL RELEASE_POOL (@SYSTEM_POOL, EVENT_POINTER);

99   2      END PROC_D;
        $EJECT
```

```
                /*****************************************************************
                **                                                              **
                **  PROZEDURNAME :  PROC_E                                      **
                **                                                              **
                **  FUNKTION     :  BEARBEITUNG DER TIMER_NACHRICHT 'T_STATISTIK'**
                **                                                              **
                **  PARAMETER    :  KEINE                                       **
                **                                                              **
                *****************************************************************/

100   1       PROC_E: PROCEDURE;

101   2        CALL STATISTIK_AUSGEBEN;

102   2        CALL TIMER_ASSIGN (STATISTIK_ZEIT, T_STATISTIK, NOT_USED);

103   2        CALL RELEASE_POOL (@SYSTEM_POOL, EVENT_POINTER);

104   2       END PROC_E;



105   1       END UEBERWACHUNG;



MODULE INFORMATION:

     CODE AREA SIZE     = 0162H    354D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0005H      5D
     MAXIMUM STACK SIZE = 000AH     10D
     427 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

Stichwortverzeichnis

Abwärtszähler 154
Adreßerweiterung 187
Änderungsfreundlichkeit 194
Anwenderschicht 32
Apple 40
Array 256
Assembler 18
Ausgabeunterbrechung 23
BACKWARD_LINK 102
Basepointer 72
Benutzerschicht 28
Benutzerschnittstelle 119
Beschreibungstafel 201
Betriebsmittel 18
Betriebssystem - Prozedur 63, 119
Betriebssystem - Schale 18
Betriebssystem - Initialisierung 78, 136
Betriebssystem - Überwachung 113
Black Box 205
Briefkastenprinzip 80
Bus - Lock 182
Bus - arbiter 182
Bus - belastung 178
Bus - erweiterung 187
Bus - zuteiler 187
Bus - zyklen 183
Busy Waiting 184
CAMAC-Bus 185
CANCEL_CLOCK 112
CASE-Verzweigung 242
CCITT 233
Centronics 42
CLOCK_QUEUE 110
Commodore 40
Compiler 24
CONTINUE 90, 129
CP/M 40
CREATE_PROCESS 51
Data Base Manager 219
Dateibehandlung 224
Datensichtstation 20

Datenübertragung 151
Datenübertragungsprotokoll 223
DBM 219
DELAY 90, 129
DELETE_PROCESS 53
DEQUEUE 122
Deskriptor 201
Detailspezifikation 190
DMA 174
Dual Port RAM 181
Dummy-Prozeß 35
Durchsatz 226
Echtzeituhr 109
Editor 24
Eigenentwicklung 191
Ein-Chip-Computer 151
Eingabeinterrupt 23
Eingangswarteschlange 201
Einketten 225
Einplatinencomputer 164
ELSE-Zweig 244
Ende-zu-Ende-Schicht 28
Endlosschliefe 167
ENQUEUE 122
Entkopplung 198
Entprellen 34
Erprobungsphase 239
EVENT_BLOCK 80
EVENT_QUEUE 53, 80
EVENT_TYPE 62
Exception Handling 163
Fehler - behandlung 163
Fehler - routine 248
Fehler - suche 164
Fehler - zustände 165
Fehlertolerante Systeme 188
FIFO 54, 119
Firmware 151
FIRST_IN_QUEUE 65, 121
Floppy Disk 20
Flußdiagramme 242
FORWARD_LINK 68
FORWARD_LINK 121
Funktionsteilung 223

GET_POOL 63, 124
GET_POOL_DIS 124
Grobspezifikation 190
Hardware-Anreize 225
Hardware-Entwicklung 38
Hardware-Ereignisse 38
Hardware-Timer-Interrupt 98, 138
Hierarchieebenen 203
Hierarchische Zergliederung 203
iAPX 86/30 96, 144
IBM Personal Computer 40
Idle-Prozeß 86, 140
IEC-Bus 42
Information Hiding 202
Initialisierungsteil 256
Inkonsistenzen 219
Integration 202, 205
Internmeldung 265
Interrupt 158
Interrupt-Handler 214
Interrupt-Prozeß 214
Interruptprogramm 197
Interruptsperre 214
Interrupttabelle 136
iRMX 86 144
ISO 32
JOBCODE 259
Kassettenrecorder 40
Kommerzielles Betriebssystem 191
Kosten 198
Kritische Region 85
Last 188
Last - spitzen 151
Last - abwehr 206
Last - situation 166
Last - teilung 224
Last - verbund 188
LAST_IN_QUEUE 65, 121
Last Look 34
Laufzeit 212
LED 165
Lesezugriff 221
LIFO 56
Linker 24

LOCK	184
Logikanalysator	38
Lose Kopplung	29
Magnetband	20
Massenspeicher	40
MDS	141
Mehrdimensionaler CASE	242
Mehrrechnersysteme	29
Meldungsfluß	211
Minicomputer	179
Minimal - datenspeicher	154
Minimal - systeme	154
Minimum Mode	173
Modul	47
Modulare Programmierung	47
Modularisierung	194
Modularität	198
Monitorfunktionen	38
Monitorprogramm	24
MTOS 86	48, 145
Nachladbare Prozesse	50
Netzschicht	32
NMI	162
Non maskable Interrupt	162
Normung	32
NULL_ID	65
Nullzeichen	23
NUMBER_IN_QUEUE	65, 136
Offenes Kommunikationssystem	28,
Online-Tracer	37
OWNER_PID	62
Parallel-Serien-Wandlung	21
Pegelstände	211
Peripherie - bausteine	158
Personal Computer	40
Pflichtenheft	18
Physikalische Schicht	32
PID	258
Plattenspeicher	20
PL/M-86	141
Pool	60
Pool -element	60
Pool -elementkopf	62
Pool -verwaltung	60

Präsentationsschicht		32
Priorität		86, 126
PROCESS_CONTROL_BLOCK		90, 91
PROCESS_LOAD_TABLE		137
PROCESS_QUEUE		53, 119
PROCESS_SCHEDULER		92
PROM, EPROM		151
Prozedur		47
Prozedur	- parameter	220
Prozessor	- Flag	163
Prozessor	- belastung	166
Prozessor	- status	178
Prozessor	- takt	157
Prozeß		47
Prozeß	- aufbau	201
Prozeß	- kommunikation	79, 126
Prozeß	- rahmen	257
Prozeß	- start	78
Prozeß	- synchronisation	86
Prozeß	- wechsel	75
Prozeß	- zustände	51
Pufferspeicher		196
Queue		53, 119
QUEUE_HEADER		64, 120,152
Rahmenprogramm		248
Rahmenspezifikation		190
RAM	- bedarf	151
RAM	- Speicher	151
RAM-Speicher		41
REAL_TIME_PROCESS		86
REAL_TIME_PROCESS		119
RECEIVE_EVENT		83, 126
Rechner-Kommunikationssystem		25
Rechnernetzwerk		32
RELEASE_POOL		63, 124
RELEASE_POOL_DIS		125
RESET_PRIORITY		90, 130
RMOS		43, 145
RMS 68 K		146
ROM		44
ROSY		47, 139, 147
Round Robin Scheduling		93
SBC-System		171
Schachtelung (Interrupts)		165

Schachtelungen (Programm) 245
Scheduler 48
Scheduling-Algorithmus 46, 212
Scheduling-Methoden 92
Schichten 28
Schnittstellen-Review 205
Schnittstellenformate 190
Schrittweise Verfeinerung 190
Semaphore 85
Semaphore-Mechanismus 220
SEND_EVENT 83, 126
Sensor 158
Sensor-Information 204
Sensor-Änderung 204
Sensormeldungen 197
Serielle Schnittstelle 21
SET 244
SET_CLOCK 110
SET_PRIORITY 90, 130
SIGNAL_INTERRUPT 90, 129
Sitzungsschicht 32
SME 141
SMP-System 43
Software - Interrupt 162
Software - Verriegelung 182
Software-Entwicklung 38
Software-Ereignisse 38
Softwarekosten 40
SOURCE_PID 62
Spannungs - Reset 168
Spannungs - einbruch 162
Spannungs - versorgung 162
Speicher - Pool 152
Speicher - block 60
Speicher - fehler 166
Speicher - lokaler 181
Speicher - medium 164
Speicher - physikalisch 164
Speicher - test 166
Speicher - zellen 165
Speicher - zugriff, direkter 174
Sprachsyntax 244
Stack 56
Stack - mechanismus 56

Stack	- pointer	72
Stack	- register	93
Stack	- segment	72, 141
STACK_ARRAY		78
START_PROCESS		51
Struktogramm		119
Strukturelemente		244
Strukturierte Sprache		245
SUSPEND		84
System	- bus	181
System	- dynamik	216
System	- kalender	162
System	- konfiguration	221
System	- konzept	175
System	- speicher	181
System	- start	135
Systemintegration		35
Tabellenzugriffe		201
Task-Multiplexer		48
Tastenscanner		34
Teilnehmerindividuelle Organisation		256
Telefoncomputer		17
Teste-und-Setze-Funktion		183
Texteditor		45
Time-Slot		98
TIME_SLOT_POINTER		98
TIME_SLOT_TABLE		53, 98
Timer		94
TIMER_ASSIGN		94
TIMER_QUEUE		53
TIMER_RELEASE		95
TIMER_TYPE		80, 131
Timer-Tabelle		152
Timer-Zeit		152
Timesharing-Betrieb		93
Timesharing-Betriebssystem		42
Tintenstrahldrucker		40
Top-Down		190, 233
Transportschicht		28, 32
Überwachungskonzept		239
Überwachungsprozesse		256
Uhr		226
Uhrenprogramm		162
UNIX		42

UNQUEUE 119
UNSUSPEND 83, 92
Unterbrechung 158
Unterbrechungs - ebene 164
Unterbrechungs - leitung 175
Unterbrechungs - steuerung 19
Unterbrechungsanforderung 21
USER_CODE 80
Verbergen von Information 205
Verbindungsschicht 32
Verriegelung 198
Vertagen von Meldungen 51
Verteilte Systeme 223
Verteiltes Betriebssystem 182
Virtuelle Adressierung 163
Virtuelle Maschine 48
WAIT_INTERRUPT 84, 129
Wait-State-Generator 179
Wartbarkeit 194
Warten, beschäftigtes 179
Warteschlange 53
Warteschlangenfunktionen 38
Warteschlangenkopf 64
Wartezyklen 179
Wechselseitiger Ausschluß 220
Wettkampfanzeige 194
Winchester-Plattenlaufwerke 40
X.21 32 42
X.25 32
Zähler 155
Zähler - baustein 156
Zählrate 156
Zeit - auftrag 157
Zeit - basis 155
Zeit - geber 152
Zeit - glieder 94, 131
Zeit - interrupt 175
Zeit - raster 155
Zeit - takt 155
Zeit - verhalten 175
Zeit - verwaltung 93
Zeit - zähler 152
Zeitauflösung 195
Zeitkritische Prozesse 256

Zeitliche Zergliederung 204
Zergliederung 194
ZRTS 8000 148
Zugriffsberechtigung 220
Zugriffsprozedur 220
Zugriffsversuch 220
Zustands - diagramme 203
Zustands - -Anreiz-Tabelle 240
Zustands - programmierung 239
Zustands - tabelle 256
Zustands - übergänge 245
Zyklischer Prozeß 257

Lieferbare Markt & Technik-Titel:

CP/M und WordStar Anwenderhandbuch
Best.-Nr. MT 310 DM 29,80*

Software-Auswahl leicht gemacht
Best.-Nr. MT 340 DM 58,—*

Hardware-Auswahl leicht gemacht
3. überarbeitete und aktualisierte Ausgabe 1984/85
Best.-Nr. MT 350 DM 58,—*

Personal Computer Lexikon
Best.-Nr. MT 390 DM 19,80*

Planen und kalkulieren mit VisiCalc
Best.-Nr. MT 450 DM 32,—*

Basic ohne Probleme:
Bd. 1 Unterweisung
Best.-Nr. MT 480 DM 36,—*

Basic ohne Probleme:
Bd. 2 Übungen
Best.-Nr. MT 490 DM 26,—*

Basic ohne Probleme:
Bd. 3 Programmentwicklung und Datenverwaltung
Best.-Nr. MT 500 DM 44,—*

Basic ohne Probleme:
Bd. 4 Allgemeine Dateiverwaltung am praktischen Beispiel
Best.-Nr. MT 514 DM 53,—*

Basic-Programme für CBM/VC-20-Computer
Best.-Nr. MT 501 DM 32,—*

Planen und kalkulieren mit Multiplan
Best.-Nr. MT 502 DM 58,—*

Der IBM-Personal Computer
Best.-Nr. MT 503 DM 53,—*

Datenkommunikation und Lokale Computer-Netzwerke
Best.-Nr. MT 504 DM 58,—*

Software richtig eingekauft
Best.-Nr. MT 505 DM 34,—*

Wörterbuch der Daten- und Telekommunikation
Best.-Nr. MT 506 DM 38,—*

Multiplan richtig eingesetzt
Best.-Nr. MT 507 DM 58,—*

Multiplan richtig eingesetzt — Beispiele auf Diskette (5¼", IBM-PC mit MS-DOS 2.0)
Best.-Nr. MT 623 DM 48,—*

Personal Computer — das intelligente Werkzeug für jedermann
Best.-Nr. MT 508 DM 53,—*

SuperCalc richtig eingesetzt
Best.-Nr. MT 511 DM 58,—*

SuperCalc richtig eingesetzt — Beispiele auf Diskette (5¼", IBM-PC mit MS-DOS 2.0)
Best.-Nr. MT 621 DM 48,—*

Programme und Tips für VC-20
Best.-Nr. MT 513 DM 38,—*

Das VC-20 Buch
Best.-Nr. MT 516 DM 49,—*

Das VC-20 Buch — Beispiele auf Kassette
Best.-Nr. MT 581 DM 19,90*

Das VC-20 Buch — Beispiele auf Diskette
Best.-Nr. MT 582 DM 29,90*

Ein-Chip-Mikrocomputer-Handbuch
Best.-Nr. MT 517 DM 58,—*

Die Btx-Fibel
Best.-Nr. MT 519 DM 29,80*

Das Datenbanksystem dBASE II
Best.-Nr. MT 524 DM 68,—*

Basic-80 und CP/M
Best.-Nr. MT 525 DM 48,—*

Einführung in Datenbanksysteme mit dBASE II
Best.-Nr. MT 526 DM 68,—*

Einführung in Datenbanksysteme mit dBASE II — Beispiele auf Diskette (5¼", IBM-PC mit MS-DOS 2.0)
Best.-Nr. MT 622 DM 48,—*

dBASE II richtig eingesetzt
Best.-Nr. MT 541 DM 68,—*

dBASE II richtig eingesetzt — Beispiele auf Diskette (5¼", IBM-PC mit MS-DOS 2.0))
Best.-Nr. MT 544 DM 48,—*

Einführung in C
Best.-Nr. MT 561 DM 69,—*

Mit Lotus 1-2-3 zur integrierten Problemlösung
Best.-Nr. MT 562 DM 68,—*

Mit Lotus 1-2-3 zur integrierten Problemlösung (Beispiele auf Diskette)
Best.-Nr. MT 647 DM 58,—*

* inkl. MwSt. zuzügl. Versandkosten

Markt&Technik

Hans-Pinsel-Straße 2 · 8013 Haar bei München · Telefon 4613-220

Lieferbare Markt & Technik-Titel:

Basic-Dialekte im Vergleich
Best.-Nr. MT 564 — DM 32,—*

Das Commodore 64-Buch, Bd. 1: Leitfaden für Erstanwender — mit Assembler
Best.-Nr. MT 591 — DM 48,—*

Das Commodore 64-Buch, Bd. 1: Beispiele auf Diskette
Best.-Nr. MT 592 — DM 58,—*

Das Commodore 64-Buch, Bd. 2: Basic-Spiele
Best.-Nr. MT 593 — DM 38,—*

Das Commodore 64-Buch, Bd. 2: Beispiele auf Diskette
Best.-Nr. MT 594 — DM 58,—*

Das Commodore 64-Buch, Bd. 3: Leitfaden für Fortgeschrittene
Best.-Nr. MT 595 — DM 38,—*

Das Commodore 64-Buch, Bd. 3: Beispiele auf Diskette
Best.-Nr. MT 596 — DM 58,—*

Das Commodore 64-Buch, Bd. 4: Leitfaden für Programmierer — Assembler — Disassembler
Best.-Nr. MT 597 — DM 38,—*

Das Commodore 64-Buch, Bd. 4: Beispiele auf Diskette
Best.-Nr. MT 598 — DM 58,—*

Das Commodore 64-Buch, Bd. 5: Simon's Basic
Best.-Nr. MT 599 — DM 38,—*

Das Commodore 64-Buch, Bd. 5: Beispiele auf Diskette
Best.-Nr. MT 600 — DM 58,—*

Das Commodore 64-Buch, Bd. 6: Spiele
Best.-Nr. MT 619 — DM 38,—*

Das Commodore 64-Buch, Bd. 6: Beispiele auf Diskette
Best.-Nr. MT 620 — DM 58,—*

Computerspiele und Wissenswertes — Commodore 64
Best.-Nr. MT 601 — DM 29,80*

Computerspiele und Wissenswertes — Commodore 64: Beispiele auf Diskette
Best.-Nr. MT 602 — DM 38,—*

Das große Spielebuch Commodore 64
Best.-Nr. MT 603 — DM 29,80*

Das große Spielebuch Commodore 64: Beispiele auf Diskette
Best.-Nr. MT 604 — DM 38,—*

Software-Schnellkurs: CP/M
Best.-Nr. MT 605 — DM 37,—*

Software-Schnellkurs: MailMerge
Best.-Nr. MT 606 — DM 37,—*

Software-Schnellkurs: dBASE II
Best.-Nr. MT 607 — DM 37,—*

Software-Schnellkurs: SuperCalc
Best.-Nr. MT 608 — DM 37,—*

Software-Schnellkurs: WordStar
Best.-Nr. MT 609 — DM 37,—*

Software-Schnellkurs: Multiplan
Best.-Nr. MT 610 — DM 37,—*

Software-Schnellkurs: Lotus 1-2-3
Best.-Nr. MT 611 — DM 48,—*

Software-Schnellkurs: CP/M 86
Best.-Nr. MT 615 — DM 37,—*

Software-Schnellkurs: MS-DOS
Best.-Nr. MT 651 — DM 37,—*

Mehr als 32 Basic-Programme für den Commodore 64
Best.-Nr. MT 613 — DM 49,—*

Mehr als 32 Basic-Programme für den Commodore 64: Beispiele auf Diskette
Best.-Nr. MT 614 — DM 48,—*

Mehr als 32 Basic-Programme für den IBM-PC
Best.-Nr. MT 624 — DM 68,—*

Mehr als 32 Basic-Programme für den IBM-PC: Beispiele auf Diskette (5¼" mit MS-DOS 2.0)
Best.-Nr. MT 625 — DM 58,—*

MS-DOS
Best.-Nr. MT 616 — DM 58,—*

Einführung in Forth
Best.-Nr. 635 — DM 58,—*

Lotus 1-2-3 richtig eingesetzt
Best.-Nr. MT 637 — DM 68,—*

Lotus 1-2-3 richtig eingesetzt Beispiele auf Diskette (5¼", IBM-PC mit MS-DOS 2.0)
Best.-Nr. MT 638 — DM 58,—*

Logo — Grafik, Sprache, Mathematik
Best.-Nr. MT 648 — DM 42,—*

WordStar für die Praxis
Best.-Nr. MT 642 — DM 54,—*

* Inkl. MwSt. zuzügl. Versandkosten

Markt&Technik

Hans-Pinsel-Straße 2 · 8013 Haar bei München · Telefon 4613-220

Echtzeit-Betriebssysteme für Mikrocomputer

Realzeitanwendungen stellen bei Mikrocomputersystemen besondere Anforderungen an Hardware, Systemsoftware und Anwendersoftware. Das Buch beschreibt den Stand der Technik für den Entwurf und die Realisierung von Realzeitsystemen. Den Schwerpunkt bilden Echtzeit-Betriebssysteme und wie man mit ihnen in der Praxis umgeht. Im Anhang ist ein kompletter Betriebssystemkern abgedruckt, an dem Schlagworte wie »Prozeßkonzept«, »Scheduling«, »Warteschlangen«, »Timer« usw. ausführlich erläutert werden. Breiter Raum ist dem Systementwurf in Hardware und besonders in Software gewidmet, und zwar immer im Hinblick auf den Einsatz eines Realzeit-Betriebssystems. Der letzte Teil des Buches bildet eine praktische Anleitung für das Erstellen von Anwenderprogrammen. Alle Kapitel sind reichlich mit Beispielen und Bildern versehen.

DR. CLEMENS SCHMIDT

geboren 1947 in Überlingen am Bodensee. Studium der Physik an der Universität Tübingen. Promotion und Wissenschaftlicher Mitarbeiter am Max-Planck-Institut für Festkörperforschung in Stuttgart. Dazwischen Lehr- und Forschungsaufenthalt in Argentinien. Seit 1980 Mitarbeiter bei der Siemens AG, auf dem Gebiet Systementwicklung für neue Kommunikationssysteme tätig.

DIETMAR ALBRECHT

geboren 1954 in Stuttgart. Studium der Nachrichtentechnik an der Fachhochschule Ulm. Seit 1978 Mitarbeiter der Siemens AG, in der Hardware-Entwicklung auf dem Gebiet der PCM-Technik und seit 1980 mit der Software-Entwicklung für Systeme der Breitband-Kommunikationstechnik (BIGFON) beschäftigt.

Markt&Technik Verlag Aktiengesellschaft
ISBN 3-922120-84-9